



iSeries

DB2 Universal Database for iSeries SQL Programming Concepts

Version 5





@server

iSeries

DB2 Universal Database for iSeries SQL Programming Concepts

Version 5

Contents

About DB2 UDB for iSeries SQL

Programming Concepts. ix

Who should read this book	ix
Assumptions relating to examples of SQL statements	ix
Code disclaimer information	x
How to interpret syntax diagrams in this guide.	x
What's new in the V5R2 version of the SQL programming concepts information	xii

Chapter 1. Introduction to DB2 UDB for iSeries Structured Query Language. 1

SQL concepts	1
SQL relational database and system terminology	3
Types of SQL statements	4
SQL Communication Area (SQLCA)	6
SQL objects	6
Schemas	6
Tables, Rows, and Columns	7
Aliases	7
Views.	7
Indexes	8
Constraints	8
Triggers	8
Stored Procedures	9
User-defined functions	9
User-defined types	9
SQL Packages	9
Application program objects.	10
User source file member	11
Output source file member	11
Program	12
SQL Package	12
Module.	12
Service program	13

Chapter 2. Getting Started with SQL 15

Starting interactive SQL	15
Creating a schema	16
Example: Creating the schema (SAMPLECOLL)	16
Creating and using a table	16
Example: Creating a table (INVENTORY_LIST)	17
Creating the Supplier Table (SUPPLIERS)	18
Using the LABEL ON statement	18
Inserting information into a table	20
Example: Inserting information into a table (INVENTORY_LIST)	20
Getting information from a single table	22
Getting information from more than one table.	25
Changing information in a table	27
Example: Changing information in a table	27
Deleting information from a table	29
Example: Deleting information from a table (INVENTORY_LIST)	29
Creating and using a view	30

Example: Creating a view on a single table.	31
Example: Creating a view combining data from more than one table	31

Chapter 3. Getting started with iSeries Navigator Database 33

Starting iSeries Navigator.	33
Creating a library with iSeries Navigator	33
Example: Creating a library using iSeries Navigator (SAMPLELIB)	34
Edit list of libraries displayed in iSeries Navigator	35
Creating and using a table using iSeries Navigator	35
Example: Creating a table (INVENTORY_LIST) using iSeries Navigator	36
Defining columns on a table using iSeries Navigator	36
Creating the supplier table (SUPPLIERS) using iSeries Navigator	38
Copying column definitions using iSeries Navigator	38
Inserting information into a table using iSeries Navigator	38
Viewing the contents of a table using iSeries Navigator	40
Changing information in a table using iSeries Navigator	40
Deleting information from a table using iSeries Navigator	40
Copying and moving a table using iSeries Navigator	41
Creating and using a view with iSeries Navigator	42
Creating a view over a single table using iSeries Navigator	42
Creating a view combining data from more than one table using iSeries Navigator	44
Deleting database objects using iSeries Navigator.	46

Chapter 4. Data Definition Language (DDL) 47

Creating a schema	47
Creating a table	48
Adding constraints to a table	48
Creating a table using LIKE	48
Creating a table using AS.	49
Declaring a global temporary table	49
Creating and altering an identity column	50
ROWID.	51
Creating descriptive labels using the LABEL ON statement	51
Describing an SQL object using COMMENT ON	52
Getting comments after running a COMMENT ON statement	52
Changing a table definition	52
Adding a column	53

Changing a column	53
Allowable conversions	53
Deleting a column	54
Order of operations for ALTER TABLE statement	54
Creating and using ALIAS names	55
Creating and using views	55
Creating a view with UNION	57
Adding indexes	57
Catalogs in database design	58
Getting catalog information about a table	58
Getting catalog information about a column	58
Dropping a database object	59

Chapter 5. Retrieving data using the SELECT statement 61

Querying data using the SELECT statement	61
Specifying a search condition using the WHERE clause	63
Expressions in the WHERE Clause	63
Comparison operators	65
NOT Keyword	65
GROUP BY clause	65
HAVING clause	67
ORDER BY clause	68
Static SELECT statements	70
Null Values to indicate absence of column values in a row	71
Special registers in SQL statements	71
Casting data types	72
Date, Time, and Timestamp data types	73
Specifying current date and time values	73
Date/Time arithmetic	74
Preventing duplicate rows	74
Performing complex search conditions	75
Special considerations for LIKE	76
Multiple search conditions within a WHERE clause	77
Joining data from more than one table	78
Inner Join	79
Left Outer Join	80
Right Outer Join	80
Exception Join	81
Cross Join	81
Simulating a Full Outer Join	82
Multiple join types in one statement	82
Using table expressions	83
Using the UNION keyword to combine subselects	85
Specifying UNION ALL	88
Data retrieval errors	89

Chapter 6. SQL Insert, Update, and Delete 93

Inserting rows using the INSERT statement	93
Inserting rows into a table using a Select-Statement	95
Inserting multiple rows in a table with the blocked INSERT statement	96
Inserting into an identity column	97
Changing data in a table using the UPDATE statement	97

Updating a table using a scalar-subselect	99
Updating a table with rows from another table	99
Updating an identity column	100
Updating data as it is retrieved from a table	100
Removing rows from a table using the DELETE statement	102

Chapter 7. Using Subqueries. 105

Subqueries in SELECT statements	105
Correlation	106
Subqueries and search conditions	106
How subqueries are used	107
Notes on using subqueries	108
Correlated subqueries	109
Correlated names and references	109
Example: Correlated subquery in a WHERE Clause	110
Example: Correlated subquery in a HAVING Clause	111
Example: Correlated subquery in select-list	112
Using correlated subqueries in an UPDATE statement	112
Using correlated subqueries in a DELETE statement	113

Chapter 8. Sort sequences in SQL . . . 115

Sort sequence used with ORDER BY and row selection	115
Sort sequence and ORDER BY	116
Row selection	117
Sort sequence and views	118
Sort Sequence and the CREATE INDEX Statement	119
Sort sequence and constraints	119

Chapter 9. Using a Cursor 121

Types of cursors	121
Serial cursor	121
Scrollable cursor	122
Example of using a cursor	122
Step 1: Define the cursor	124
Step 2: Open the cursor	126
Step 3: Specify what to do when end-of-data is reached	126
Step 4: Retrieve a row using a cursor	126
Step 5a: Update the current row	127
Step 5b: Delete the current row	127
Step 6: Close the cursor	128
Using the multiple-row FETCH statement	128
Multiple-row FETCH using a host structure array	129
Multiple-row FETCH using a row storage area	131
Unit of work and open cursors	133

Chapter 10. Data Integrity 135

Adding and using check constraints	135
Referential integrity	136
Adding or dropping referential constraints	137
Removing referential constraints	138
Inserting into tables with referential constraints	138
Updating tables with referential constraints	140

Deleting from tables with referential constraints	141
Check pending	144
WITH CHECK OPTION on a View	144
WITH CASCADED CHECK OPTION	145
WITH LOCAL CHECK OPTION	146
DB2 UDB for iSeries Trigger support	147
SQL triggers	148
Creating an SQL trigger	148
BEFORE SQL triggers	149
AFTER SQL triggers	150
Handlers in SQL triggers	151
SQL trigger transition tables	152
External triggers	152
External trigger example program	152

Chapter 11. Stored Procedures 159

Defining an external procedure	160
Defining an SQL procedure	161
Debugging a stored procedure	166
Invoking a stored procedure	167
Using CALL Statement where procedure definition exists	167
Using Embedded CALL Statement where no procedure definition exists	168
Using Embedded CALL statement with an SQLDA	169
Using Dynamic CALL Statement where no CREATE PROCEDURE exists	170
Parameter passing conventions for stored procedures and UDFs	171
Indicator variables and stored procedures	176
Returning a completion status to the calling program	178
Examples of CALL statements	179
Example 1: ILE C and PL/I procedures called from ILE C applications	179
Example 2. Sample REXX Procedure Called From C Application	184

Chapter 12. Using the Object-Relational Capabilities 189

Why use the DB2 object extensions?	189
DB2 approach to supporting objects	190
Using Large Objects (LOBs)	190
Understanding large object data types (BLOB, CLOB, DBCLOB)	191
Understanding large object locators	191
Example: Using a locator to work with a CLOB value	192
Indicator variables and LOB locators	195
LOB file reference variables	195
Example: Extracting a document to a file	196
Example: Inserting data into a CLOB column	198
Display layout of LOB columns	199
Journal entry layout of LOB columns	199
User-defined functions (UDF)	200
Why use UDFs?	200
UDF concepts	203
Implementing UDFs	205
Registering UDFs	205

Save and restore considerations	206
Examples: Registering UDFs	206
Using UDFs	210
User-defined distinct types (UDT)	215
Why use UDTs?	215
Defining a UDT	216
Resolving unqualified UDTs	216
Examples: Using CREATE DISTINCT TYPE	216
Defining tables with UDTs	217
Manipulating UDTs	218
Examples of manipulating UDTs	218
Synergy between UDTs, UDFs, and LOBs	222
Combining UDTs, UDFs, and LOBs	223
Examples of complex applications	223
Using DataLinks	225
NO LINK CONTROL	227
FILE LINK CONTROL (with File System Permissions)	227
FILE LINK CONTROL (with Database Permissions)	227
Commands used for working with DataLinks	227

Chapter 13. Writing User-Defined Functions (UDFs). 231

UDF runtime environment	231
Length of time that the UDF runs	231
Threads considerations	232
Parallel processing	232
Writing function code	232
Writing UDFs as SQL functions	233
Writing UDFs as external functions	234
Examples of UDF code	242
Example: Square of a number UDF	242
Example: Counter	244
Example: Weather table function	244

Chapter 14. Dynamic SQL Applications 251

Designing and running a dynamic SQL application	253
Processing non-SELECT statements	254
CCSID of dynamic SQL statements	254
Using the PREPARE and EXECUTE statements	254
Processing SELECT statements and using an SQLDA	255
Fixed-list SELECT statements	255
Varying-list Select-statements	256
SQL Descriptor Area (SQLDA)	257
SQLDA format	257
Example: Select-statement for allocating storage for SQLDA	262
Parameter markers	267

Chapter 15. Use of dynamic SQL through client interfaces 269

Accessing data with Java	269
Accessing data with Domino	269
Accessing data with Open Database Connectivity (ODBC)	269
Accessing data with Portable Application Solutions Environment (PASE)	269

Chapter 16. Advanced database functions using iSeries Navigator . . .	271
Mapping your database using Database Navigator	271
Creating a Database Navigator map	272
Adding new objects to a map	273
Changing the objects to include in a map	273
Creating a user-defined relationship	273
Querying your database using Run SQL Scripts	274
Creating an SQL script	275
Running SQL scripts	275
Changing the options for running an SQL script	275
Viewing the result set for a stored procedure	276
Viewing the Job Log	276
Reconstructing SQL statements using Generate SQL	277
Generate SQL for database objects	277
Editing list of objects for which to generate SQL	277
Advanced table functions using iSeries Navigator	277
Creating an alias using iSeries Navigator	278
Adding indexes using iSeries Navigator	278
Adding key constraints using iSeries Navigator	279
Adding check constraints using iSeries Navigator	280
Adding referential constraints using iSeries Navigator	281
Adding triggers using iSeries Navigator	281
Enabling and disabling a trigger	282
Removing constraints and triggers	282
Defining SQL objects using iSeries Navigator	283
Defining a stored procedure using iSeries Navigator	283
Defining a user-defined function using iSeries Navigator	283
Defining a user-defined type using iSeries Navigator	284
Creating an SQL Package	284
Chapter 17. Using Interactive SQL . . .	285
Basic functions of interactive SQL	285
Starting interactive SQL	286
Using statement entry function	287
Prompting	288
Using the list selection function	290
Session services description	293
Exiting interactive SQL	294
Using an existing SQL session	295
Recovering an SQL session	295
Accessing remote databases with interactive SQL	295
Chapter 18. Using the SQL Statement Processor	299
Execution of statements after errors occur	300
Commitment control in the SQL statement processor	300
Schemas in the SQL Statement Processor	300
Source member listing for the SQL statement processor	301
Chapter 19. DB2 UDB for iSeries Data Protection	305

Security for SQL objects	305
Authorization ID	306
Views	306
Auditing	306
Securing data using iSeries Navigator	307
Defining public authority for an object	307
Setting up default public authority for new objects	307
Authorizing a user or group to an object	307
Data integrity	308
Concurrency	308
Journaling	310
Commitment control	311
Savepoints	314
Atomic operations	316
Constraints	317
Save/Restore	318
Damage tolerance	319
Index recovery	319
Catalog integrity	320
User auxiliary storage pool (ASP)	320
Independent auxiliary storage pool (IASP)	321

Chapter 20. Testing SQL Statements in Application Programs. 323

Establishing a test environment	323
Designing a test data structure	323
Testing your SQL application programs	324
Program debug phase	324
Performance verification phase	325

Chapter 21. Distributed Relational Database Function 327

DB2 UDB for iSeries distributed relational database support	328
DB2 UDB for iSeries distributed relational database example program	328
SQL package support	329
Valid SQL statements in an SQL package	330
Considerations for creating an SQL package	330
CCSID considerations for SQL	333
Connection management and activation groups	334
Connections and conversations	334
Source code for PGM1:	335
Source code for PGM2:	335
Source code for PGM3:	335
Multiple connections to the same relational database	337
Implicit connection management for the default activation group	338
Implicit connection management for nondefault activation groups	339
Distributed support	339
Determining connection type	340
Connect and commitment control restrictions	343
Determining connection status	343
Distributed unit of work connection considerations	345
Ending connections	346
Distributed unit of work	346

Managing distributed unit of work connections	347
Cursors and prepared statements	349
Application requester driver programs	350
Problem handling	351
DRDA stored procedure considerations	351

Appendix A. DB2 UDB for iSeries

Sample Tables 353

Department Table (DEPARTMENT)	354
DEPARTMENT	354
Employee Table (EMPLOYEE)	355
EMPLOYEE	357
Employee Photo Table (EMP_PHOTO)	357
EMP_PHOTO	358
Employee Resume Table (EMP_RESUME)	358
EMP_RESUME	359
Employee to Project Activity Table (EMPPROJACT)	359
EMPPROJACT	360
Project Table (PROJECT)	362
PROJECT	363
Project Activity Table (PROJACT)	364
PROJACT	364
Activity Table (ACT)	366
ACT	366
Class Schedule Table (CL_SCHED)	367
CL_SCHED	367

In Tray Table (IN_TRAY)	368
IN_TRAY	368
Organization Table (ORG)	369
ORG	369
Staff Table (STAFF)	370
STAFF	370
Sales Table (SALES)	371
SALES	372

Appendix B. DB2 UDB for iSeries CL

Command Descriptions 375

CRTSQPKG (Create Structured Query Language Package) Command	375
DLTSQPKG (Delete Structured Query Language Package) Command	379
PRTSQLINF (Print Structured Query Language Information) Command	380
RUNSQLSTM (Run Structured Query Language Statement) Command	381
STRSQL (Start Structured Query Language) Command	391

Bibliography 399

Index 401

About DB2 UDB for iSeries SQL Programming Concepts

This book explains basic SQL programming concepts that show programmers and database administrators:

- How to use the DB2 UDB for iSeries licensed program
- How to access data in a database
- How to prepare, run, and test an application program that contains SQL statements.

For more information on DB2 UDB for iSeries SQL guidelines and examples for implementation in an application programming environment, see the following books in the iSeries Information Center.

- SQL Reference
- DB2 UDB for iSeries SQL Programming for Host Languages
- DB2 UDB for iSeries Database Performance and Query Optimization
- SQL Call Level Interface (ODBC)
- SQL Messages and Codes

For more information about this guide, see the following topics:

- “Who should read this book”
- “Assumptions relating to examples of SQL statements”
- “Code disclaimer information” on page x
- “How to interpret syntax diagrams in this guide” on page x
- “What’s new in the V5R2 version of the SQL programming concepts information” on page xii

Who should read this book

This guide should be used by application programmers and database administrators who are familiar with and can program with COBOL for iSeries, ILE COBOL for iSeries, iSeries PL/I, ILE C for iSeries, ILE C++, REXX, RPG III (part of RPG for iSeries), or ILE RPG for iSeries language and who can understand basic database applications.

Also see the following sections:

- “Assumptions relating to examples of SQL statements”
- “Code disclaimer information” on page x
- “How to interpret syntax diagrams in this guide” on page x

Assumptions relating to examples of SQL statements

The examples of SQL statements shown in this guide are based on the sample tables in Appendix A, DB2 UDB for iSeries Sample Tables, and assume the following:

- They are shown in the interactive SQL environment or they are written in ILE C or in COBOL. EXEC SQL and END-EXEC are used to delimit an SQL statement in a COBOL program. A description of how to use SQL statements in a COBOL program is provided in “Coding SQL Statements in COBOL Applications.” A

description of how to use SQL statements in an ILE C program is provided in "Coding SQL Statements in C Applications."

- Each SQL example is shown on several lines, with each clause of the statement on a separate line.
- SQL keywords are highlighted.
- Table names provided in Appendix A, DB2 UDB for iSeries Sample Tables, use the schema CORPDATA. Table names that are not found in the Sample Tables should use schemas you create.
- Calculated columns are enclosed in parentheses, (), and brackets, [].
- The SQL naming convention is used.
- The APOST and APOSTSQL precompiler options are assumed although they are not the default options in COBOL. Character string literals within SQL and host language statements are delimited by apostrophes (').
- A sort sequence of *HEX is used, unless otherwise noted.
- The complete syntax of the SQL statement is usually not shown in any one example. For the complete description and syntax of any of the statements described in this guide, see the SQL Reference

Whenever the examples vary from these assumptions, it is stated.

Because this guide is for the application programmer, most of the examples are shown as if they were written in an application program. However, many examples can be slightly changed and run interactively by using interactive SQL. The syntax of an SQL statement, when using interactive SQL, differs slightly from the format of the same statement when it is embedded in a program.

Code disclaimer information

This document contains programming examples.

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

All sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

All programs contained herein are provided to you "AS IS" without any warranties of any kind. The implied warranties of non-infringement, merchantability and fitness for a particular purpose are expressly disclaimed.

How to interpret syntax diagrams in this guide

Throughout this book, syntax is described using the structure defined as follows:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The ►— symbol indicates the beginning of a statement.

The —> symbol indicates that the statement syntax is continued on the next line.

The ►— symbol indicates that a statement is continued from the previous line.

The —>◀ symbol indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the \blacktriangleright symbol and end with the \blacktriangleright symbol.

- Required items appear on the horizontal line (the main path).



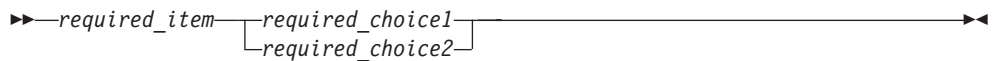
- Optional items appear below the main path.



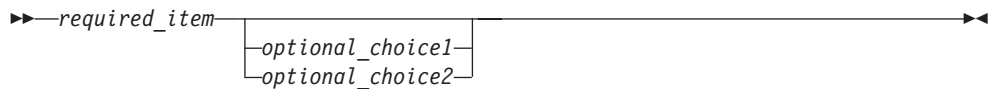
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.



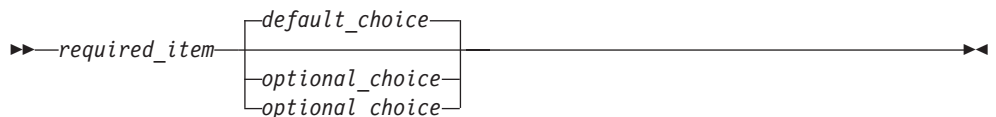
- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.



If choosing one of the items is optional, the entire stack appears below the main path.



If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.



If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

What's new in the V5R2 version of the SQL programming concepts information

The major changes to this information for this release were:

- User Defined Table Functions. See "User-defined functions (UDF)" on page 200 for details.
- Unfenced User-Defined functions. See "Making a function fenced or unfenced" on page 242 for details.
- Savepoints. See "Savepoints" on page 314 for details.
- Temporary tables
- Identity columns. See "Creating and altering an identity column" on page 50 for details.
- Scalar subselects
- Debugging SQL procedures. "Debugging a stored procedure" on page 166 for details.

Chapter 1. Introduction to DB2 UDB for iSeries Structured Query Language

These topics describe the iSeries server implementation of the Structured Query Language (SQL) using DB2 UDB for iSeries and the DB2 UDB Query Manager and SQL Development Kit Version 5 licensed program. SQL manages information based on the relational model of data. SQL statements can be embedded in high-level languages, dynamically prepared and run, or run interactively.

SQL consists of statements and clauses that describe what you want to do with the data in a database and under what conditions you want to do it.

This topic describes the following:

- “SQL concepts”
- “SQL objects” on page 6
- “Application program objects” on page 10

SQL can access data in a remote relational database, using the IBM Distributed Relational Database Architecture* (DRDA*). This function is described in the Chapter 21, “Distributed Relational Database Function” topic in this guide. Further information about DRDA is contained in the Distributed Database Programming book.

SQL concepts

DB2 UDB for iSeries SQL consists of the following main parts:

- SQL run-time support

SQL run-time parses SQL statements and runs any SQL statements. This support is that part of the Operating System/400* (OS/400) licensed program which allows applications that contain SQL statements to be run on systems where the DB2 UDB Query Manager and SQL Development Kit licensed program is not installed.

- SQL precompilers

SQL precompilers support precompiling embedded SQL statements in host languages. The following languages are supported:

- ILE C
- ILE C++ for iSeries
- ILE COBOL
- COBOL for iSeries
- iSeries PL/I
- RPG III (part of RPG for iSeries)
- ILE RPG

The SQL host language precompilers prepare an application program containing SQL statements. The host language compilers then compile the precompiled host source programs. For more information about precompiling, see the topic Preparing and Running a Program with SQL Statements in the *SQL Programming with Host Languages* information. The precompiler support is part of the DB2 UDB Query Manager and SQL Development Kit licensed program.

- SQL interactive interface
SQL interactive interface allows you to create and run SQL statements. For more information about interactive SQL, see Chapter 17, “Using Interactive SQL”. Interactive SQL is part of the DB2 UDB Query Manager and SQL Development Kit licensed program.
- Run SQL Scripts
The Run SQL Scripts window in iSeries Navigator allows you to create, edit, run, and troubleshoot scripts of SQL statements. Run SQL Scripts is a part of iSeries Navigator. For more information, see “Querying your database using Run SQL Scripts” on page 274.
- Run SQL Statements CL command
RUNSQLSTM allows you to run a series of SQL statements, which are stored in a source file. See Chapter 18, “Using the SQL Statement Processor” for more information about the Run SQL Statements command.
- DB2 Query Manager for iSeries
DB2 Query Manager for iSeries provides a prompt-driven interactive interface that allows you to create data, add data, maintain data, and run reports on the databases. Query Manager is part of the DB2 UDB Query Manager and SQL Development Kit licensed program. For more information, refer to the Query Manager Use book.
- SQL REXX Interface
The SQL REXX interface allows you to run SQL statements in a REXX procedure. For more information about using SQL statements in REXX procedures, see the topic Coding SQL Statements in REXX Applications in the *SQL Programming with Host Languages* information.
- SQL Call Level Interface
DB2 UDB for iSeries supports the SQL Call Level Interface. This allows users of any of the ILE languages to access SQL functions directly through procedure calls to a service program provided by the system. Using the SQL Call Level Interface, one can perform all the SQL functions without the need for a precompile. This is a standard set of procedure calls to prepare SQL statements, execute SQL statements, fetch rows of data, and even do advanced functions such as accessing the catalogs and binding program variables to output columns. For a complete description of all the available functions, and their syntax, see the SQL Call Level Interface (ODBC) book.
- QSQPRCED API
This Application Program Interface (API) provides an extended dynamic SQL capability. SQL statements can be prepared into an SQL package and then executed using this API. Statements prepared into a package by this API persist until the package or statement is explicitly dropped. QSQPRCED is part of the OS/400 licensed program. For more information about the QSQPRCED API, see the QSQPRCED topic in the Programming section of the iSeries Information Center. For general information on APIs, see the OS/400 API topic in the iSeries Information Center.
- QSQCHKS API
This API syntax checks SQL statements. QSQCHKS is part of the OS/400 licensed program. For more information about the QSQCHKS API, see the QSQCHKS topic in the Programming section of the iSeries Information Center. For general information on APIs, see the OS/400 API topic in the iSeries Information Center.
- DB2 Multisystem

This feature of the operating system allows your data to be distributed across multiple servers. For more information about DB2 Multisystem, see the DB2 Multisystem book.

- DB2 UDB Symmetric Multiprocessing

This feature of the operating system provides the query optimizer with additional methods for retrieving data that include parallel processing. Symmetric multiprocessing (SMP) is a form of parallelism achieved on a single system where multiple processors (CPU and I/O processors) that share memory and disk resource work simultaneously towards achieving a single end result. This parallel processing means that the database manager can have more than one (or all) of the system processors working on a single query simultaneously. See the topic Controlling Parallel Processing in the *Database Performance and Query Optimization* information for details on how to control parallel processing.

For more information, see the following sections:

- “SQL relational database and system terminology”
- “Types of SQL statements” on page 4
- “SQL Communication Area (SQLCA)” on page 6

SQL relational database and system terminology

In the relational model of data, all data is perceived as existing in tables. DB2 UDB for iSeries objects are created and maintained as system objects. The following table shows the relationship between system terms and SQL relational database terms. For more information about database programming using the traditional file interface, see the Database Programming book.

Table 1. Relationship of System Terms to SQL Terms

System Terms	SQL Terms
Library. Groups related objects and allows you to find the objects by name.	Schema. Consists of a library, a journal, a journal receiver, an SQL catalog, and optionally a data dictionary. A schema groups related objects and allows you to find the objects by name.
Physical file. A set of records.	Table. A set of columns and rows.
Record. A set of fields.	Row. The horizontal part of a table containing a serial set of columns.
Field. One or more characters of related information of one data type.	Column. The vertical part of a table of one data type.
Logical file. A subset of fields and records of one or more physical files.	View. A subset of columns and rows of one or more tables.
SQL Package. An object type that is used to run SQL statements.	Package. An object type that is used to run SQL statements.
User Profile	Authorization name or Authorization ID.

See also:

- “SQL and system naming conventions”

SQL and system naming conventions

There are two naming conventions that can be used in DB2 UDB for iSeries programming: system (*SYS) and SQL (*SQL). The naming convention used affects the method for qualifying file and table names and the terms used on the interactive SQL displays. The naming convention used is selected by a parameter

on the SQL commands or, for REXX, selected through the SET OPTION statement. See Qualification of unqualified object names in the SQL Reference for more details.

System naming (*SYS): In the system naming convention, tables and other SQL objects in an SQL statement are qualified by schema name in the form:
schema/table

SQL naming (*SQL): In the SQL naming convention, tables and other SQL objects in an SQL statement are qualified by the schema name in the form:
schema.table

Types of SQL statements

There are four basic types of SQL statements:

- Data definition language (DDL) statements
- Data manipulation language (DML) statements
- Dynamic SQL statements
- Miscellaneous statements

SQL statements can operate on objects that are created by SQL as well as externally described physical files and single-format logical files, whether or not they reside in an SQL schema. They do not refer to the IDDU dictionary definition for program-described files. Program-described files appear as a table with only a single column.

SQL DDL Statements

ALTER TABLE
COMMENT ON
CREATE ALIAS
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE INDEX
CREATE PROCEDURE
CREATE SCHEMA
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DECLARE GLOBAL TEMPORARY TABLE
DROP ALIAS
DROP DISTINCT TYPE
DROP FUNCTION
DROP INDEX
DROP PACKAGE
DROP PROCEDURE
DROP SCHEMA
DROP TABLE
DROP TRIGGER
DROP VIEW
GRANT DISTINCT TYPE
GRANT FUNCTION
GRANT PACKAGE
GRANT PROCEDURE
GRANT TABLE
LABEL ON
RENAME
REVOKE DISTINCT TYPE
REVOKE FUNCTION
REVOKE PACKAGE
REVOKE PROCEDURE
REVOKE TABLE

Dynamic SQL Statements

DESCRIBE
EXECUTE
EXECUTE IMMEDIATE
PREPARE

SQL DML Statements

CLOSE
COMMIT
DECLARE CURSOR
DELETE
FETCH
INSERT
LOCK TABLE
OPEN
RELEASE SAVEPOINT
ROLLBACK
SAVEPOINT
SELECT INTO
SET variable
UPDATE
VALUES INTO

Miscellaneous Statements

BEGIN DECLARE SECTION
CALL
CONNECT
DECLARE PROCEDURE
DECLARE STATEMENT
DECLARE VARIABLE
DESCRIBE TABLE
DISCONNECT
END DECLARE SECTION
FREE LOCATOR
HOLD LOCATOR
INCLUDE
RELEASE
SET CONNECTION
SET OPTION
SET PATH
SET RESULT SETS
SET SCHEMA
SET TRANSACTION
WHENEVER

SQL DDL statements are described in Chapter 4, "Data Definition Language (DDL)" on page 47. SQL DML statements are described in Chapter 5, "Retrieving data using the SELECT statement" on page 61 and Chapter 6, "SQL Insert, Update, and Delete" on page 93. You can find complete descriptions of these statements in the SQL Reference book.

SQL Communication Area (SQLCA)

An SQLCA is a set of variables that is updated at the end of the execution of every SQL statement. A program that contains executable SQL statements must provide exactly one SQLCA (unless a stand-alone SQLCODE or a stand-alone SQLSTATE variable is used instead). For more information, see SQL Communication Area topic in the *SQL Reference* book in iSeries Information Center.

SQL objects

SQL objects are schemas, data dictionaries, journals, catalogs, tables, aliases, views, indexes, constraints, triggers, stored procedures, user-defined functions, user-defined types, and SQL packages. SQL creates and maintains these objects as system objects. A brief description of these objects follows:

- "Schemas"
- "Data Dictionary"
- "Journals and Journal Receivers" on page 7
- "Catalogs" on page 7
- "Tables, Rows, and Columns" on page 7
- "Aliases" on page 7
- "Views" on page 7
- "Indexes" on page 8
- "Constraints" on page 8
- "Triggers" on page 8
- "Stored Procedures" on page 9
- "User-defined functions" on page 9
- "User-defined types" on page 9
- "SQL Packages" on page 9

Schemas

A schema provides a logical grouping of SQL objects. A **schema** consists of a library, a journal, a journal receiver, a catalog, and optionally, a data dictionary. Tables, views, and system objects (such as programs) can be created, moved, or restored into any system library. All system files can be created or moved into an SQL schema if the SQL schema does not contain a data dictionary. If the SQL schema contains a data dictionary then:


- Source physical files or nonsource physical files with one member can be created, moved, or restored into an SQL schema.
- Logical files cannot be placed in an SQL schema because they cannot be described in the data dictionary.

You can create and own many schemas. The term *collection* can be used synonymously with *schema*.

Data Dictionary

A schema contains a data dictionary if it was created prior to Version 3 Release 1 or if the WITH DATA DICTIONARY clause was specified on the CREATE

SCHEMA statements. A **data dictionary** is a set of tables containing object definitions. If SQL created the dictionary, then it is automatically maintained by the system. You can work with data dictionaries by using the interactive data definition utility (IDDU), which is part of the OS/400 program. For more

information about IDDU, see the IDDU Use  book.

Journals and Journal Receivers

A **journal** and **journal receiver** are used to record changes to tables and views in the database. The journal and journal receiver are then used in processing SQL COMMIT, ROLLBACK, SAVEPOINT, and RELEASE SAVEPOINT statements. The journal and journal receiver can also be used as an audit trail or for forward or backward recovery. For more information about journaling, see the Journaling topic. For more information about commitment control, see the Commitment control topic.

Catalogs

An SQL **catalog** consists of a set of tables and views which describe tables, views, indexes, packages, procedures, functions, files, triggers, and constraints. This information is contained in a set of cross-reference tables in libraries QSYS and QSYS2. In each SQL schema there is a set of views built over the catalog tables that contains information about the tables, views, indexes, packages, files, and constraints in the schema.

A catalog is automatically created when you create a schema. You cannot drop or explicitly change the catalog.

For more information about SQL catalogs, see the Catalogs topic in the SQL Reference book.

Tables, Rows, and Columns

A **table** is a two-dimensional arrangement of data consisting of **rows** and **columns**. The row is the horizontal part containing one or more columns. The column is the vertical part containing one or more rows of data of one data type. All data for a column must be of the same type. A table in SQL is a keyed or nonkeyed physical file. See the Data types topic in the SQL Reference book for a description of data types.

Data in a table can be distributed across servers. For more information about distributed tables, see the DB2 Multisystem book.

Aliases

An **alias** is an alternate name for a table or view. You can use an alias to refer to a table or view in those cases where an existing table or view can be referred to. Additionally, aliases can be used to join table members. For more information about aliases, see the Alias topic in the SQL Reference book.

Views

A **view** appears like a table to an application program; however, a view contains no data. It is created over one or more tables. A view can contain all the columns of given tables or some subset of them, and can contain all the rows of given tables or some subset of them. The columns can be arranged differently in a view than they are in the tables from which they are taken. A view in SQL is a special form of a nonkeyed logical file.

For more information about views, see Views in the *SQL Reference* book in the iSeries Information Center.

Indexes

An SQL **index** is a subset of the data in the columns of a table that are logically arranged in either ascending or descending order. Each index contains a separate arrangement. These arrangements are used for ordering (ORDER BY clause), grouping (GROUP BY clause), and joining. An SQL index is a keyed logical file.

The index is used by the system for faster data retrieval. Creating an index is optional. You can create any number of indexes. You can create or drop an index at any time. The index is automatically maintained by the system. However, because the indexes are maintained by the system, a large number of indexes can adversely affect the performance of applications that change the table.

For more information about coding effective indexes, see Using indexes to speed access to large tables topic in the *Database Performance and Query Optimization* book in the iSeries Information Center.

Constraints

Constraints are rules enforced by the database manager. DB2 UDB for iSeries supports the following constraints:

- Unique constraints

A **unique constraint** is the rule that the values of the key are valid only if they are unique. Unique constraints can be created using the CREATE TABLE and ALTER TABLE statements. Although CREATE INDEX can create a unique index that also guarantees uniqueness, such an index is not a constraint.

Unique constraints are enforced during the execution of INSERT and UPDATE statements. A PRIMARY KEY constraint is a form of UNIQUE constraint. The difference is that a PRIMARY KEY cannot contain any nullable columns.

- Referential constraints

A **referential constraint** is the rule that the values of the foreign key are valid only if:

- They appear as values of a parent key, or
- Some component of the foreign key is null.

Referential constraints are enforced during the execution of INSERT, UPDATE, and DELETE statements.

- Check constraints

A **check constraint** is a rule that limits the values allowed in a column or group of columns. Check constraints can be added using the CREATE TABLE and ALTER TABLE statements. Check constraints are enforced during the execution of INSERT and UPDATE statements. To satisfy the constraint, each row of data inserted or updated in the table must make the specified condition either TRUE or unknown (due to a null value).

For more information about constraints, see Chapter 10, “Data Integrity”.

Triggers

A **trigger** is a set of actions that are executed automatically whenever a specified event occurs to a specified base table. An event can be an insert, update, delete, or read operation. The trigger can be run either before or after the event. DB2 UDB for iSeries supports SQL insert, update, and delete triggers and external triggers.

For more information about triggers, see Chapter 10, “Data Integrity” in this book or see the Triggering automatic events in your database topic in the *Database Programming* book.

Stored Procedures

A **stored procedure** is a program that can be called using the SQL CALL statement. DB2 UDB for iSeries supports external stored procedures and SQL procedures. External stored procedures can be any system program or REXX procedure. They cannot be System/36 programs or procedures, or service programs. An SQL procedure is defined entirely in SQL and can contain SQL statements including SQL control statements. For more information about stored procedures, see the Chapter 11, “Stored Procedures” topic in this book.

User-defined functions

A **user-defined function** is a program that can be invoked like any built-in function. DB2 UDB for iSeries supports external functions, SQL functions, and sourced functions. External functions can be any system ILE program or service program. An SQL function is defined entirely in SQL and can contain SQL statements, including SQL control statements. A sourced function is built over any built-in or any existing user-defined function. You can create a scalar function or a table function as either an SQL or external function. For more information about user-defined functions, see the Chapter 13, “Writing User-Defined Functions (UDFs)” on page 231.

User-defined types

A **user-defined type** is a distinct data type that users can define independently of those supplied by the database management system. Distinct data types map on a one-to-one basis to existing database types. For more information about user-defined types, see the “User-defined distinct types (UDT)” on page 215.

SQL Packages

An SQL package is an object that contains the control structure produced when the SQL statements in an application program are bound to a remote relational database management system (DBMS). The DBMS uses the control structure to process SQL statements encountered while running the application program.

SQL packages are created when a relational database name (RDB parameter) is specified on a Create SQL (CRTSQLxxx) command and a program object is created. Packages can also be created using the CRTSQLPKG command. For more information about packages and distributed relational database function, see Chapter 21, “Distributed Relational Database Function”.

SQL packages can also be created using the QSQPRCED API. The references to SQL Packages within this book refer exclusively to Distributed Program SQL packages. QSQPRCED uses SQL Packages to provide Extended Dynamic SQL support. For more information about QSQPRCED, see the QSQPRCED topic in the OS/400 API section of the iSeries Information Center.

Note: The xxx in this command refers to the host language indicators: CI for the ILE C language, CPPI for the ILE C++ for iSeries language, CBL for the COBOL for iSeries language, CBLI for the ILE COBOL language, PLI for the iSeries PL/I language, RPG for the RPG for iSeries language, and RPGI for the ILE RPG language.

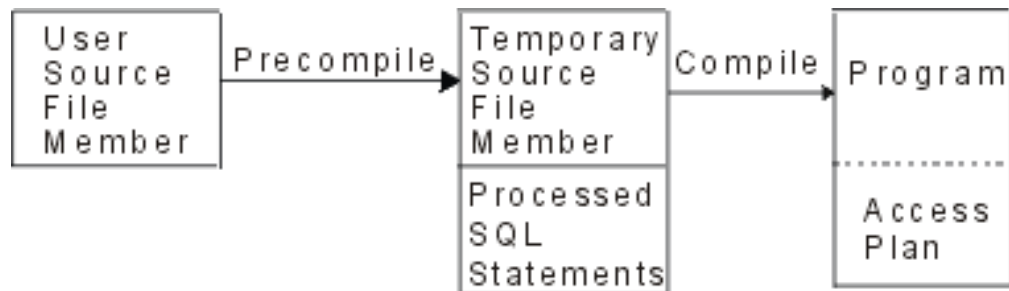
Application program objects

The process of creating a DB2 UDB for iSeries application program may result in the creation of several objects. This section briefly describes the process of creating a DB2 UDB for iSeries application. DB2 UDB for iSeries supports both non-ILE and ILE precompilers. Application programs may be either distributed or nondistributed. Additional information on creating DB2 UDB for iSeries application programs is in the topic *Preparing and Running a Program with SQL Statements* in the *SQL Programming with Host Languages* information.

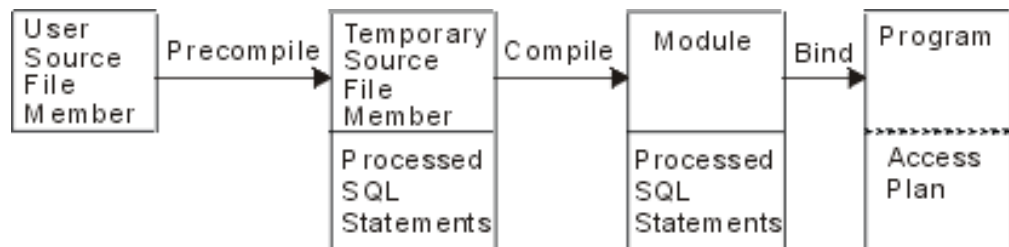
With DB2 UDB for iSeries you may need to manage the following objects:

- The original source
- Optionally, the module object for ILE programs
- The program or service program
- The SQL package for distributed programs

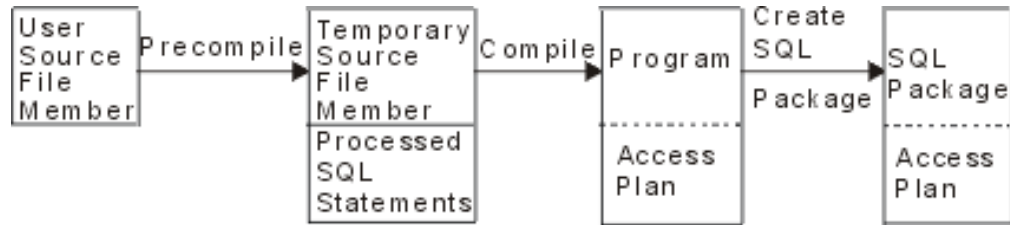
With a nondistributed non-ILE DB2 UDB for iSeries program, you must manage only the original source and the resulting program. The following shows the objects involved and steps that happen during the precompile and compile processes for a nondistributed non-ILE DB2 UDB for iSeries program:



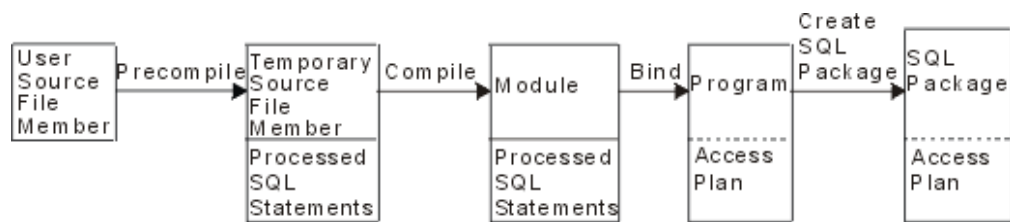
With a nondistributed ILE DB2 UDB for iSeries program, you may need to manage the original source, the modules, and the resulting program or service program. The following shows the objects involved and steps that happen during the precompile and compile processes for a nondistributed ILE DB2 UDB for iSeries program when OBJTYPE(*PGM) is specified on the precompile command:



With a distributed non-ILE DB2 UDB for iSeries program, you must manage the original source, the resulting program, and the resulting package. The following shows the objects and steps that occur during the precompile and compile processes for a distributed non-ILE DB2 UDB for iSeries program:



With a distributed ILE DB2 UDB for iSeries program, you must manage the original source, module objects, the resulting program or service program, and the resulting packages. An SQL package can be created for each distributed module in a distributed ILE program or service program. The following shows the objects and steps that occur during the precompile and compile processes for a distributed ILE DB2 UDB for iSeries program:



Note: The access plans associated with the DB2 UDB for iSeries distributed program object are not created until the program is run locally.

For more information, see the following sections:

- “User source file member”
- “Output source file member”
- “Program” on page 12
- “Module” on page 12
- “Service program” on page 13

User source file member

A source file member contains the programmer’s application language and SQL statements. You can create and maintain the source file member by using the source entry utility (SEU), a part of the IBM WebSphere Development Studio for iSeries licensed program.

Output source file member

The SQL precompile creates an output source file member. By default, the precompile process creates a temporary source file QSQLTxxxxx in QTEMP, or you can specify the output source file as a permanent file name on the precompile command. If the precompile process uses the QTEMP library, the system automatically deletes the file when the job completes. A member with the same name as the program name is added to the output source file. This member contains the following items:

- Calls to the SQL run-time support, which have replaced embedded SQL statements
- Parsed and syntax-checked SQL statements

By default, the precompiler calls the host language compiler. For more information about precompilers, see the topic *Preparing and Running a Program with SQL Statements* in the *SQL Programming with Host Languages* information.

Program

A program is the object which you can run that is created as a result of the compile process for non-ILE compiles or as a result of the bind process for ILE compiles.

An access plan is a set of internal structures and information that tells SQL how to run an embedded SQL statement most effectively. It is created only when the program has successfully created. Access plans are not created during program creation for SQL statements if the statements:

- Refer to a table or view that cannot be found
- Refer to a table or view to which you are not authorized

The access plans for such statements are created when the program is run. If, at that time, the table or view still cannot be found or you are still not authorized, a negative SQLCODE is returned. Access plans are stored and maintained in the program object for nondistributed SQL programs and in the SQL package for distributed SQL programs.

SQL Package

An SQL package contains the access plans for a distributed SQL program.

An SQL package is an object that is created when:

- A distributed SQL program is successfully created using the RDB parameter on CRTSQLxxx commands.
- When the Create SQL Package (CRTSQLPKG) command is run.

When a distributed SQL program is created, the name of the SQL package and an internal consistency token are saved in the program. These are used at run time to find the SQL package and to verify that the SQL package is correct for this program. Because the name of the SQL package is critical for running distributed SQL programs, an SQL package cannot be:

- Moved
- Renamed
- Duplicated
- Restored to a different library

Module

A module is an Integrated Language Environment (ILE) object that is created by compiling source code using the CRTxxxMOD command (or any of the CRTBNDxxx commands where xxx is C, CBL, CPP, or RPG). You can run a module only if you use the Create Program (CRTPGM) command to bind it into a program. You usually bind several modules together, but you can bind a module by itself. Modules contain information about the SQL statements; however, the SQL access plans are not created until the modules are bound into either a program or service program. See the Create Program (CRTPGM) in the Command Language topic for more information about Create Program (CRTPGM).

Service program

A service program is an Integrated Language Environment (ILE) object that provides a means of packaging externally supported callable routines (functions or procedures) into a separate object. Bound programs and other service programs can access these routines by resolving their imports to the exports provided by a service program. The connections to these services are made when the calling programs are created. This improves call performance to these routines without including the code in the calling program.

Chapter 2. Getting Started with SQL

This chapter describes how to create and work with schemas, tables, and views using SQL statements in Interactive SQL.

The syntax for each of the SQL statements used in this chapter is described in detail in the SQL Reference book. A description of how to use SQL statements and clauses in more complex situations is provided in Chapter 4, “Data Definition Language (DDL)” on page 47, Chapter 5, “Retrieving data using the SELECT statement” and Chapter 6, “SQL Insert, Update, and Delete”.

In this chapter, the examples use the interactive SQL interface to show the execution of SQL statements. Each SQL interface provides methods for using SQL statements to define tables, views, and other objects, methods for updating the objects, and methods for reading data from the objects.

See the following topics for details:

- “Starting interactive SQL”
- “Creating a schema” on page 16
- “Creating and using a table” on page 16
- “Using the LABEL ON statement” on page 18
- “Inserting information into a table” on page 20
- “Getting information from a single table” on page 22
- “Getting information from more than one table” on page 25
- “Changing information in a table” on page 27
- “Deleting information from a table” on page 29
- “Creating and using a view” on page 30

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Starting interactive SQL

To start using interactive SQL for the following examples, type:

```
STRSQL NAMING(*SQL)
```

and press Enter. When the Enter SQL Statements display appears, you are ready to start typing SQL Statements. For more information about interactive SQL and the STRSQL command, see Chapter 17, “Using Interactive SQL”.

If you are reusing an existing interactive SQL session, make sure that you set the naming mode to *SQL naming*. You can specify this on the F13 (Services) panel, option 1 (Change session attributes).

Creating a schema

A schema is the basic object in which tables, views, indexes, and packages are placed. For more information about creating a schema, see SQL CREATE SCHEMA statement.

Note: The term *collection* can be used synonymously with *schema*.

For an example of creating a schema using interactive SQL, see “Example: Creating the schema (SAMPLECOLL)”.

Example: Creating the schema (SAMPLECOLL)

You can create a sample schema, named SAMPLECOLL, by typing the following SQL statement on the Enter SQL Statements display and pressing Enter:

```
Enter SQL Statements

Type SQL statement, press Enter.
Current connection is to relational database SYSTEM1
====> CREATE SCHEMA SAMPLECOLL
_____
_____
_____
Bottom
F3=Exit  F4=Prompt  F6=Insert line  F9=Retrieve  F10=Copy line
F12=Cancel  F13=Services  F24=More keys
```

Note: Running this statement causes several objects to be created and takes several seconds.

After you have successfully created a schema, you can create tables, views, and indexes in it. Tables, views, and indexes can also be created in libraries instead of schemas.

Creating and using a table

You can create a table by using the SQL CREATE TABLE statement. The CREATE TABLE statement allows you to create a table, define the physical attributes of the columns in the table, and define constraints to restrict the values that are allowed in the table.

For an example of creating a table using interactive SQL, see “Example: Creating a table (INVENTORY_LIST)” on page 17.

When creating a table, you need to understand the concepts of null value and default value. A null value indicates the absence of a column value for a row. It is not the same as a value of zero or all blanks. It means “unknown”. It is not equal to any value, not even to other null values. If a column does not allow the null value, a value must be assigned to the column, either a default value or a user supplied value.

A default value is assigned to a column when a row is added to a table and no value is specified for that column. If a specific default value is not defined for a column, the system default value will be used. For more information about the default values used by INSERT, see “Inserting rows using the INSERT statement” on page 93

Example: Creating a table (INVENTORY_LIST)

We are going to create a table to maintain information about the current inventory of a business. It will have information about the items kept in the inventory, their cost, quantity currently on hand, the last order date, and the number last ordered. The item number will be a required value. It cannot be null. The item name, quantity on hand, and order quantity will have user supplied default values. The last order date and quantity ordered will allow the null value.

On the Enter SQL Statements display, type CREATE TABLE and press F4 (Prompt). The following display is shown (with the input areas not yet filled in):

```

Specify CREATE TABLE Statement

Type information, press Enter.

Table . . . . . INVENTORY_LIST _____ Name
Collection . . . . . SAMPLECOLL _____ Name, F4 for list

Nulls: 1=NULL, 2=NOT NULL, 3=NOT NULL WITH DEFAULT

Column          FOR Column  Type          Length  Scale  Nulls
ITEM_NUMBER     _____  CHAR          6       ___   2
ITEM_NAME       _____  VARCHAR       20      ___   3
UNIT_COST       _____  DECIMAL       8       2     3
QUANTITY_ON_HAND _____  SMALLINT      _____  ___   1
LAST_ORDER_DATE _____  DATE          _____  ___   1
ORDER_QUANTITY  _____  SMALLINT      _____  ___   1
_____         _____  _____     _____  ___   3
                                                    Bottom

Table CONSTRAINT . . . . . N      Y=Yes, N=No
Distributed Table . . . . . N      Y=Yes, N=No

F3=Exit   F4=Prompt   F5=Refresh  F6=Insert line  F10=Copy line
F11=Display more attributes  F12=Cancel  F14=Delete line  F24=More keys

```

Type the table name and schema name of the table you are creating, INVENTORY_LIST in SAMPLECOLL, for the *Table* and *Collection* prompts. Each column you want to define for the table is represented by an entry in the list on the lower part of the display. For each column, type the name of the column, the data type of the column, its length and scale, and the null attribute.

Press F11 to see more attributes that can be specified for the columns. This is where a default value may be specified.

Specify CREATE TABLE Statement

Type information, press Enter.

Table INVENTORY_LIST_____ Name
Collection SAMPLECOLL_____ Name, F4 for list

Data: 1=BIT, 2=SBCS, 3=MIXED, 4=CCSID

Column	Data	Allocate	CCSID	CONSTRAINT	Default
ITEM_NUMBER_____	-	_____	_____	N	_____
ITEM_NAME_____	-	_____	_____	N	'***UNKNOWN***'_____
UNIT_COST_____	-	_____	_____	N	_____
QUANTITY_ON_HAND_____	-	_____	_____	N	NULL_____
LAST_ORDER_DATE_____	-	_____	_____	N	_____
ORDER_QUANTITY_____	-	_____	_____	N	20_____
_____	-	_____	_____	-	_____

Bottom

Table CONSTRAINT N Y=Yes, N=No
Distributed Table N Y=Yes, N=No

F3=Exit F4=Prompt F5=Refresh F6=Insert line F10=Copy line
F11=Display more attributes F12=Cancel F14=Delete line F24=More keys

Note: Another way of entering column definitions is to press F4 (Prompt) with your cursor on one of the column entries in the list. This will bring up a display that shows all of the attributes for defining a single column.

When all the values have been entered, press Enter to create the table. The Enter SQL Statements display will be shown again with a message indicating that the table has been created.

You can directly key in this CREATE TABLE statement on the Enter SQL Statements display as follows:

```
CREATE TABLE SAMPLECOLL.INVENTORY_LIST
  (ITEM_NUMBER CHAR(6) NOT NULL,
   ITEM_NAME VARCHAR(20) NOT NULL WITH DEFAULT '***UNKNOWN***',
   UNIT_COST DECIMAL(8,2) NOT NULL WITH DEFAULT,
   QUANTITY_ON_HAND SMALLINT DEFAULT NULL,
   LAST_ORDER_DATE DATE,
   ORDER_QUANTITY SMALLINT DEFAULT 20)
```

Creating the Supplier Table (SUPPLIERS)

Later in our examples, we will need a second table as well. This table will contain information about suppliers of our inventory items, which items they supply, and the cost of the item from that supplier. To create it, either type it in directly on the Enter SQL Statements display or press F4 (Prompt) to use the interactive SQL displays to create the definition.

```
CREATE TABLE SAMPLECOLL.SUPPLIERS
  (SUPPLIER_NUMBER CHAR(4) NOT NULL,
   ITEM_NUMBER CHAR(6) NOT NULL,
   SUPPLIER_COST DECIMAL(8,2))
```

Using the LABEL ON statement

Normally, the column name is used as the column heading when showing the output of a SELECT statement in interactive SQL. By using the LABEL ON statement, you can create a more descriptive label for the column name. Since we are going to be running our examples in interactive SQL, we will use the LABEL ON statement to change the column headings. Even though the column names are

descriptive, it will be easier to read if the column headings show each part of the name on a single line. It will also allow us to see more columns of our data on a single display.

To change the labels for our columns, type LABEL ON COLUMN on the Enter SQL Statements display and press F4 (Prompt). The following display will appear:

```

Specify LABEL ON Statement

Type choices, press Enter.

Label on . . . . 2                                1=Table or view
                                                    2=Column
                                                    3=Package
                                                    4=Alias

Table or view      INVENTORY_LIST_____        Name, F4 for list
Collection . .    SAMPLECOLL__                  Name, F4 for list

Option . . . . . 1                                1=Column heading
                                                    2=Text

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F20=Display full names
F21=Display statement

```

Type in the name of the table and schema containing the columns for which you want to add labels and press Enter. The following display will be shown, prompting you for each of the columns in the table.

```

Specify LABEL ON Statement

Type information, press Enter.

Column          Column Heading
.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....
ITEM_NUMBER     'ITEM          NUMBER' _____
ITEM_NAME       'ITEM          NAME' _____
UNIT_COST       'UNIT          COST' _____
QUANTITY_ON_HAND 'QUANTITY     ON           HAND' _____
LAST_ORDER_DATE 'LAST         ORDER          DATE' _____
ORDER_QUANTITY  'NUMBER       ORDERED' _____

Bottom

F3=Exit      F5=Refresh  F6=Insert line  F10=Copy line  F12=Cancel
F14=Delete line  F19=Display system column names  F24=More keys

```

Type the column headings for each of the columns. Column headings are defined in 20 character sections. Each section will be displayed on a different line when showing the output of a SELECT statement. The ruler across the top of the column heading entry area can be used to easily space the headings correctly. When the headings are typed, press Enter.

The following message indicates that the LABEL ON statement was successful.

```
LABEL ON for INVEN00001 in SAMPLECOLL completed.
```

The table name in the message is the system table name for this table, not the name that was actually specified in the statement. DB2 UDB for iSeries maintains two names for tables with names longer than ten characters. For more information about system table names, see the CREATE TABLE statement in the SQL Reference book.

The LABEL ON statement can also be keyed in directly on the Enter SQL statements display as follows:

```
LABEL ON SAMPLECOLL.INVENTORY_LIST
(ITEM_NUMBER IS 'ITEM          NUMBER',
 ITEM_NAME   IS 'ITEM          NAME',
 UNIT_COST   IS 'UNIT          COST',
 QUANTITY_ON_HAND IS 'QUANTITY ON          HAND',
 LAST_ORDER_DATE IS 'LAST      ORDER          DATE',
 ORDER_QUANTITY IS 'NUMBER    ORDERED')
```

Inserting information into a table

After you create a table, you can insert, or add, information (data) into it by using the SQL INSERT statement.

For an example of inserting data into a table using interactive SQL, see “Example: Inserting information into a table (INVENTORY_LIST)”.

Example: Inserting information into a table (INVENTORY_LIST)

To work with interactive SQL, on the Enter SQL Statements display, type INSERT and press F4 (Prompt). The Specify INSERT Statement display will be shown.

Specify INSERT Statement

Type choices, press Enter.

INTO table	INVENTORY_LIST _____	Name, F4 for list
Collection	SAMPLECOLL_	Name, F4 for list

Select columns to insert

INTO	Y	Y=Yes, N=No
Insertion method	1	1=Input VALUES 2=Subselect

Type choices, press Enter.

WITH isolation level . . .	1	1=Current level, 2=NC (NONE) 3=UR (CHG), 4=CS, 5=RS (ALL) 6=RR
----------------------------	---	--

F3=Exit F4=Prompt F5=Refresh F12=Cancel F20=Display full names
F21=Display statement

Type the table name and schema name in the input fields as shown. Change the *Select columns to insert INTO* prompt to Yes. Press Enter to see the display where the columns you want to insert values into can be selected.

Specify INSERT Statement

Type sequence numbers (1-999) to make selections, press Enter.

Seq	Column	Type	Length	Scale
1	ITEM_NUMBER	CHARACTER	6	
2	ITEM_NAME	VARCHAR	20	
3	UNIT_COST	DECIMAL	8	2
4	QUANTITY_ON_HAND	SMALLINT	4	
	LAST_ORDER_DATE	DATE		
	ORDER_QUANTITY	SMALLINT	4	

Bottom

F3=Exit F5=Refresh F12=Cancel F19=Display system column names
 F20=Display entire name F21=Display statement

In this example, we only want to insert into four of the columns. We will let the other columns have their default value inserted. The sequence numbers on this display indicate the order that the columns and values will be listed in the INSERT statement. Press Enter to show the display where values for the selected columns can be typed.

Specify INSERT Statement

Type values to insert, press Enter.

Column	Value
ITEM_NUMBER	'153047'
ITEM_NAME	'Pencils, red'
UNIT_COST	10.00
QUANTITY_ON_HAND	25

Bottom

F3=Exit F5=Refresh F6=Insert line F10=Copy line F11=Display type
 F12=Cancel F14=Delete line F15=Split line F24=More keys

Note: To see the data type and length for each of the columns in the insert list, press F11 (Display type). This will show a different view of the insert values display, providing information about the column definition.

Type the values to be inserted for all of the columns and press Enter. A row containing these values will be added to the table. The values for the columns that were not specified will have a default value inserted. For LAST_ORDER_DATE it will be the null value since no default was provided and the column allows the null value. For ORDER_QUANTITY it will be 20, the value specified as the default value on the CREATE TABLE statement.

You can type the INSERT statement on the Enter SQL Statements display as:

```
INSERT INTO SAMPLECOLL.INVENTORY_LIST
  (ITEM_NUMBER,
   ITEM_NAME,
   UNIT_COST,
   QUANTITY_ON_HAND)
```

```
VALUES('153047',
      'Pencils, red',
      10.00,
      25)
```

To add the next row to the table, press F9 (Retrieve) on the Enter SQL Statements display. This will copy the previous INSERT statement to the typing area. You can either type over the values from the previous INSERT statement or press F4 (Prompt) to use the Interactive SQL displays to enter data.

Continue using the INSERT statement to add the following rows to the table. Values not shown in the chart below should not be inserted so that the default will be used. In the INSERT statement column list, specify only the column names for which you want to insert a value. For example, to insert the third row, you would specify only ITEM_NUMBER and UNIT_COST for the column names and only the two values for these columns in the VALUES list.

ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND
153047	Pencils, red	10.00	25
229740	Lined tablets	1.50	120
544931		5.00	
303476	Paper clips	2.00	100
559343	Envelopes, legal	3.00	500
291124	Envelopes, standard		
775298	Chairs, secretary	225.00	6
073956	Pens, black	20.00	25

Add the following rows to the SAMPLECOLL.SUPPLIERS table.

SUPPLIER_NUMBER	ITEM_NUMBER	SUPPLIER_COST
1234	153047	10.00
1234	229740	1.00
1234	303476	3.00
9988	153047	8.00
9988	559343	3.00
2424	153047	9.00
2424	303476	2.50
5546	775298	225.00
3366	303476	1.50
3366	073956	17.00

The sample schema now contains two tables with several rows of data in each.

Getting information from a single table

Now that we have inserted all the information into our tables, we need to be able to look at it again. In SQL, this is done with the SELECT statement. The SELECT statement is the most complex of all SQL statements. This statement is composed of three main clauses:

1. The SELECT clause, which specifies those columns containing the desired data.
2. The FROM clause, which specifies the table or tables containing the columns with the desired data.
3. The WHERE clause, which supplies conditions that determine which rows of data are retrieved.

In addition to the three main clauses, there are several other clauses described in Chapter 5, "Retrieving data using the SELECT statement" on page 61, and in the SQL Reference book, that can affect the final form of returned data.

To see the values we inserted into the INVENTORY_LIST table, type SELECT and press F4 (prompt). The following display will be shown:

```

Specify SELECT Statement
Type SELECT statement information. Press F4 for a list.

FROM tables . . . . . SAMPLECOLL.INVENTORY_LIST _____
SELECT columns . . . . . * _____
WHERE conditions . . . . . _____
GROUP BY columns . . . . . _____
HAVING conditions . . . . . _____
ORDER BY columns . . . . . _____
FOR UPDATE OF columns . . . . . _____

Bottom

Type choices, press Enter.

DISTINCT rows in result table . . . . . N Y=Yes, N=No
UNION with another SELECT . . . . . N Y=Yes, N=No
Specify additional options . . . . . N Y=Yes, N=No

F3=Exit      F4=Prompt  F5=Refresh  F6=Insert line  F9=Specify subquery
F10=Copy line F12=Cancel  F14=Delete line  F15=Split line  F24=More keys

```

Type the table name in the *FROM tables* field on the display. To select all columns from the table, type * for the *SELECT columns* field on the display. Press Enter and the statement will run to select all of the data for all of the columns in the table. The following output will be shown:

```

Display Data
Position to line . . . . . Data width . . . . . : 71
.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.
Shift to column . . . . .
ITEM  ITEM          UNIT  QUANTITY  LAST  NUMBER
NUMBER NAME          COST  ON        ORDER  ORDERED
      HAND        DATE
153047 Pencils, red      10.00   25   -      20
229740 Lined tablets    1.50   120  -      20
544931 ***UNKNOWN***    5.00   -    -      20
303476 Paper clips      2.00  100  -      20
559343 Envelopes, legal  3.00  500  -      20
291124 Envelopes, standard .00   -    -      20
775298 Chairs, secretary 225.00  6   -      20
073956 Pens, black      20.00  25  -      20
***** End of data *****

F3=Exit      F12=Cancel  F19=Left    F20=Right    F21=Split

```

The column headings that were defined using the LABEL ON statement are shown. The ITEM_NAME for the third entry has the default value that was specified in the CREATE TABLE statement. The QUANTITY_ON_HAND column

has a null value for the rows where no value was inserted. The *LAST_ORDER_DATE* column contains all null values since that column is not in any of the INSERT statements and the column was not defined to have a default value. Similarly, the *ORDER_QUANTITY* column contains the default value for all rows.

This statement could be entered on the Enter SQL Statements display as:

```
SELECT *
FROM SAMPLECOLL.INVENTORY_LIST
```

To limit the number of columns returned by the SELECT statement, the columns you want to see must be specified. To restrict the number of output rows returned, the WHERE clause is used. To see only the items that cost more than 10 dollars, and only have the values for the columns *ITEM_NUMBER*, *UNIT_COST*, and *ITEM_NAME* returned, type SELECT and press F4 (Prompt). The Specify SELECT Statement display will be shown.

Specify SELECT Statement

Type SELECT statement information. Press F4 for a list.

FROM tables	SAMPLECOLL.INVENTORY_LIST
SELECT columns	ITEM_NUMBER, UNIT_COST, ITEM_NAME
WHERE conditions	UNIT_COST > 10.00
GROUP BY columns	_____
HAVING conditions	_____
ORDER BY columns	_____
FOR UPDATE OF columns	_____

Bottom

Type choices, press Enter.

DISTINCT rows in result table	N	Y=Yes, N=No
UNION with another SELECT	N	Y=Yes, N=No
Specify additional options	N	Y=Yes, N=No

F3=Exit F4=Prompt F5=Refresh F6=Insert line F9=Specify subquery
 F10=Copy line F12=Cancel F14=Delete line F15=Split line F24=More keys

Although only one line is initially shown for each prompt on the Specify SELECT Statement display, F6 (Insert line) can be used to add more lines to any of the input areas in the top part of the display. This could be used if more columns were to be entered in the *SELECT columns* list, or a longer, more complex WHERE condition were needed.

Fill in the display as shown above. When Enter is pressed, the SELECT statement is run. The following output will be seen:

Display Data

Position to line	Data width	41
.....1.....2.....3.....4.	Shift to column	

ITEM NUMBER	UNIT COST	ITEM NAME
775298	225.00	Chairs, secretary
073956	20.00	Pens, black
***** End of data *****		

F3=Exit F12=Cancel F19=Left F20=Right F21=Split

The only rows returned are those whose data values compare with the condition specified in the WHERE clause. Furthermore, the only data values returned are from the columns you explicitly specified in the SELECT clause. Data values of columns other than those explicitly identified are not returned.

This statement could have been entered on the Enter SQL Statements display as:

```
SELECT ITEM_NUMBER, UNIT_COST, ITEM_NAME
FROM SAMPLECOLL.INVENTORY_LIST
WHERE UNIT_COST > 10.00
```

Getting information from more than one table

SQL allows you to get information from columns contained in more than one table. This operation is called a *join* operation. (For a more detailed description of the join operation, see “Joining data from more than one table” on page 78). In SQL, a join operation is specified by placing the names of those tables you want to join together into the same FROM clause of a SELECT statement.

Suppose you want to see a list of all the suppliers and the item numbers and item names for their supplied items. The item name is not in the SUPPLIERS table. It is in the INVENTORY_LIST table. Using the common column, ITEM_NUMBER, we can see all three of the columns as if they were from a single table.

Whenever the same column name exists in two or more tables being joined, the column name must be qualified by the table name to specify which column is really being referenced. In this SELECT statement, the column name ITEM_NUMBER is defined in both tables so the column name needs to be qualified by the table name. If the columns had different names, there would be no confusion so qualification would not be needed.

To perform this join, the following SELECT statement can be used. Enter it by typing it directly on the Enter SQL Statements display or by prompting. If using prompting, both table names need to be typed on the FROM tables input line.

```
SELECT SUPPLIER_NUMBER, SAMPLECOLL.INVENTORY_LIST.ITEM_NUMBER, ITEM_NAME
FROM SAMPLECOLL.SUPPLIERS, SAMPLECOLL.INVENTORY_LIST
WHERE SAMPLECOLL.SUPPLIERS.ITEM_NUMBER
      = SAMPLECOLL.INVENTORY_LIST.ITEM_NUMBER
```

Another way to enter the same statement is to use a correlation name. A correlation name provides another name for a table name to use in a statement. A correlation name must be used when the table names are the same. It can be specified following each table name in the FROM list. The previous statement could be rewritten as:

```
SELECT SUPPLIER_NUMBER, Y.ITEM_NUMBER, ITEM_NAME
FROM SAMPLECOLL.SUPPLIERS X, SAMPLECOLL.INVENTORY_LIST Y
WHERE X.ITEM_NUMBER = Y.ITEM_NUMBER
```

In this example, SAMPLECOLL.SUPPLIERS is given a correlation name of X and SAMPLECOLL.INVENTORY_LIST is given a correlation name of Y. The names X and Y are then used to qualify the ITEM_NUMBER column name.

For more information about columns and correlation names, see Correlation names in the *SQL Reference* topic in the iSeries Information Center.

Running this example returns the following output:

```

                                Display Data
                                Data width . . . . . :    45
Position to line . . . . .      Shift to column . . . . .
.....+.....1.....+.....2.....+.....3.....+.....4.....+
SUPPLIER_NUMBER  ITEM      ITEM
                  NUMBER  NAME
    1234          153047  Pencils, red
    1234          229740  Lined tablets
    1234          303476  Paper clips
    9988          153047  Pencils, red
    9988          559343  Envelopes, legal
    2424          153047  Pencils, red
    2424          303476  Paper clips
    5546          775298  Chairs, secretary
    3366          303476  Paper clips
    3366          073956  Pens, black
***** End of data *****

F3=Exit      F12=Cancel      F19=Left      F20=Right      F21=Split

```

Note: Since no ORDER BY clause was specified for the query, the order of the rows returned by your query may be different.

The data values in the result table represent a composite of the data values contained in the two tables INVENTORY_LIST and SUPPLIERS. This result table contains the supplier number from the SUPPLIER table and the item number and item name from the INVENTORY_LIST table. Any item numbers that do not appear in the SUPPLIER table are not shown in this result table. The results are not guaranteed to be in any order unless the ORDER BY clause is specified for the SELECT statement. Since we did not change any column headings for the SUPPLIER table, the SUPPLIER_NUMBER column name is used as the column heading.

The following is an example of using ORDER BY to guarantee the order of the rows. The statement will first order the result table by the SUPPLIER_NUMBER column. Rows with the same value for SUPPLIER_NUMBER will be ordered by their ITEM_NUMBER.

```

SELECT SUPPLIER_NUMBER, Y.ITEM_NUMBER, ITEM_NAME
FROM SAMPLECOLL.SUPPLIERS X, SAMPLECOLL.INVENTORY_LIST Y
WHERE X.ITEM_NUMBER = Y.ITEM_NUMBER
ORDER BY SUPPLIER_NUMBER, Y.ITEM_NUMBER

```

Running the previous statement would produce the following output.

```

                                Display Data
                                Data width . . . . . :    45
Position to line . . . . .      Shift to column . . . . .
.....+.....1.....+.....2.....+.....3.....+.....4.....+
SUPPLIER_NUMBER  ITEM      ITEM
                  NUMBER  NAME
    1234          153047  Pencils, red
    1234          229740  Lined tablets
    1234          303476  Paper clips
    2424          153047  Pencils, red
    2424          303476  Paper clips
    3366          073956  Pens, black
    3366          303476  Paper clips
    5546          775298  Chairs, secretary
    9988          153047  Pencils, red
    9988          559343  Envelopes, legal
***** End of data *****

F3=Exit      F12=Cancel      F19=Left      F20=Right      F21=Split

```

Changing information in a table

You can use the SQL UPDATE statement to change the data values in some or all of the columns of a table.

For an example of changing information in a table using interactive SQL, see “Example: Changing information in a table”.

If you want to limit the number of rows being changed during a single statement execution, use the WHERE clause with the UPDATE statement. For more information see, “Changing data in a table using the UPDATE statement” on page 97. If you do not specify the WHERE clause, all of the rows in the specified table are changed. However, if you use the WHERE clause, the system changes only the rows satisfying the conditions that you specify. For more information, see “Specifying a search condition using the WHERE clause” on page 63.

Example: Changing information in a table

Suppose we want to use interactive SQL and are placing an order for more paper clips today. To update the LAST_ORDER_DATE and ORDER_QUANTITY for item number 303476, type UPDATE and press F4 (Prompt). The Specify UPDATE Statement display will be shown.

Specify UPDATE Statement

Type choices, press Enter.

Table	INVENTORY_LIST_____	Name, F4 for list
Collection	SAMPLECOLL__	Name, F4 for list
Correlation	_____	Name

F3=Exit F4=Prompt F5=Refresh F12=Cancel F20=Display full names
F21=Display statement

After typing the table name and schema name, press Enter. The display will be shown again with the list of columns in the table.

Specify UPDATE Statement

Type choices, press Enter.

Table	INVENTORY_LIST_____	Name, F4 for list
Collection	SAMPLECOLL__	Name, F4 for list
Correlation	_____	Name

Type information, press Enter.

Column	Value
ITEM_NUMBER	_____
ITEM_NAME	_____
UNIT_COST	_____
QUANTITY_ON_HAND	_____
LAST_ORDER_DATE	CURRENT DATE _____
ORDER_QUANTITY	50 _____

Bottom

F3=Exit F4=Prompt F5=Refresh F6=Insert line F10=Copy line
 F11=Display type F12=Cancel F14=Delete line F24=More keys

Specifying CURRENT DATE for a value will change the date in all the selected rows to be today's date.

After typing the values to be updated for the table, press Enter to see the display on which the WHERE condition can be specified. If a WHERE condition is not specified, all the rows in the table will be updated using the values from the previous display.

Specify UPDATE Statement

Type WHERE conditions, press Enter. Press F4 for a list.

ITEM_NUMBER = '303476' _____

Bottom

Type choices, press Enter.

WITH isolation level . . . 1	1=Current level, 2=NC (NONE) 3=UR (CHG), 4=CS, 5=RS (ALL) 6=RR
------------------------------	--

F3=Exit F4=Prompt F5=Refresh F6=Insert line F9=Specify subquery
 F10=Copy line F12=Cancel F14=Delete line F15=Split line F24=More keys

After typing the condition, press Enter to perform the update on the table. A message will indicate that the function is complete.

This statement could have been typed on the Enter SQL Statements display as:

```

UPDATE SAMPLECOLL.INVENTORY_LIST
SET LAST_ORDER_DATE = CURRENT DATE,
    ORDER_QUANTITY = 50
WHERE ITEM_NUMBER = '303476'

```

Running a SELECT statement to get all the rows from the table (SELECT * FROM SAMPLECOLL.INVENTORY_LIST), returns the following result:

```

Display Data
Data width . . . . . : 71
Position to line . . . . . Shift to column . . . . .
....+....1....+....2....+....3....+....4....+....5....+....6....+....7.
ITEM  ITEM          UNIT  QUANTITY  LAST  NUMBER
NUMBER NAME          COST  ON        ORDER  ORDERED
      HAND        DATE
153047 Pencils, red      10.00   25   -      20
229740 Lined tablets    1.50   120  -      20
544931 ***UNKNOWN***    5.00   -    -      20
303476 Paper clips      2.00   100  05/30/94  50
559343 Envelopes, legal 3.00   500  -      20
291124 Envelopes, standard .00   -    -      20
775298 Chairs, secretary 225.00  6    -      20
073956 Pens, black      20.00  25   -      20
***** End of data *****
Bottom
F3=Exit      F12=Cancel   F19=Left    F20=Right   F21=Split

```

Only the entry for *Paper clips* was changed. The LAST_ORDER_DATE was changed to be the current date. This date is always the date the update is run. The NUMBER_ORDERED shows its updated value.

Deleting information from a table

You can delete data from a table by using the SQL DELETE statement. You can delete entire rows from a table when they no longer contain needed information or you can use the WHERE clause with the DELETE statement to identify rows to be deleted during a single statement execution. For more information, see “Removing rows from a table using the DELETE statement” on page 102.

For an example of deleting information in a table using interactive SQL, see “Example: Deleting information from a table (INVENTORY_LIST)”.

Example: Deleting information from a table (INVENTORY_LIST)

If we want to remove all the rows in our table that have the null value for the QUANTITY_ON_HAND column, you could enter the following statement on the Enter SQL Statements display:

```

DELETE
FROM SAMPLECOLL.INVENTORY_LIST
WHERE QUANTITY_ON_HAND IS NULL

```

To check a column for the null value, the IS NULL comparison is used. Running another SELECT statement after the delete has completed will return the following result table:

```

                                Display Data
                                Data width . . . . . : 71
                                Shift to column . . . . .
.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.
ITEM   ITEM          UNIT  QUANTITY  LAST   NUMBER
NUMBER NAME          COST   ON      ORDER  ORDERED
                                HAND    DATE
153047 Pencils, red      10.00   25     -      20
229740 Lined tablets     1.50   120    -      20
303476 Paper clips      2.00   100    05/30/94  50
559343 Envelopes, legal   3.00   500    -      20
775298 Chairs, secretary 225.00   6     -      20
073956 Pens, black      20.00   25     -      20
***** End of data *****
                                           Bottom
F3=Exit      F12=Cancel    F19=Left     F20=Right    F21=Split

```

The rows with a null value for QUANTITY_ON_HAND were deleted.

Creating and using a view

You may find that no single table contains all the information you need. You may also want to give users access to only part of the data in a table. Views provide a way to subset the table so that you deal with only the data you need. A view reduces complexity and, at the same time, restricts access.

You can create a view using the SQL CREATE VIEW statement. Using the CREATE VIEW statement, defining a view on a table is like creating a new table containing just the columns and rows you want. When your application uses a view, it cannot access rows or columns of the table that are not included in the view. However, rows that do not match the selection criteria may still be inserted through a view if the SQL WITH CHECK OPTION is not used. See Chapter 10, “Data Integrity” for more information about using WITH CHECK OPTION.

For examples of creating a view using interactive SQL, see the following:

- “Example: Creating a view on a single table” on page 31
- “Example: Creating a view combining data from more than one table” on page 31

In order to create a view you must have the proper authority to the tables or physical files on which the view is based. See the CREATE VIEW statement in the *SQL Reference* for a list of authorities needed.

If you do not specify column names in the view definition, the column names will be the same as those for the table on which the view is based.

You can make changes to a table through a view even if the view has a different number of columns or rows than the table. For INSERT, columns in the table that are not in the view must have a default value.

You can use the view as though it were a table, even though the view is totally dependent on one or more tables for data. The view has no data of its own and therefore requires no storage for the data. Because a view is derived from a table that exists in storage, when you update the view data, you are really updating data in the table. Therefore, views are automatically kept up-to-date as the tables they depend on are updated.

See “Creating and using views” on page 55 for additional information.

Example: Creating a view on a single table

The following example shows how to create a view on a single table. The view is built on the `INVENTORY_LIST` table. The table has six columns, but the view uses only three of the columns: `ITEM_NUMBER`, `LAST_ORDER_DATE`, and `QUANTITY_ON_HAND`. The order of the columns in the `SELECT` clause is the order in which they will appear in the view. The view will contain only the rows for items that were ordered in the last two weeks. The `CREATE VIEW` statement looks like this:

```
CREATE VIEW SAMPLECOLL.RECENT_ORDERS AS
SELECT ITEM_NUMBER, LAST_ORDER_DATE, QUANTITY_ON_HAND
FROM SAMPLECOLL.INVENTORY_LIST
WHERE LAST_ORDER_DATE > CURRENT DATE - 14 DAYS
```

In the example above, the columns in the view have the same name as the columns in the table because no column list follows the view name. The schema that the view is created into does not need to be the same schema as the table it is built over. Any schema or library could be used. The following display is the result of running the SQL statement:

```
SELECT * FROM SAMPLECOLL.RECENT_ORDERS
```

```
Display Data
Data width . . . . . : 26
Shift to column . . . . .
Position to line . . . . .
....+....1....+....2....+
ITEM  LAST  QUANTITY
NUMBER ORDER  ON
      DATE   HAND
303476 05/30/94  100
***** End of data *****
Bottom
F3=Exit   F12=Cancel   F19=Left   F20=Right   F21=Split
```

The only row selected by the view is the row that we updated to have the current date. All other dates in our table still have the null value so they are not returned.

Example: Creating a view combining data from more than one table

You can create a view that combines data from two or more tables by naming more than one table in the `FROM` clause. In the following example, the `INVENTORY_LIST` table contains a column of item numbers called `ITEM_NUMBER`, and a column with the cost of the item, `UNIT_COST`. These are joined with the `ITEM_NUMBER` column and the `SUPPLIER_COST` column of the `SUPPLIERS` table. A `WHERE` clause is used to limit the number of rows returned. The view will only contain those item numbers for suppliers that can supply an item at lower cost than the current unit cost.

The `CREATE VIEW` statement looks like this:

```
CREATE VIEW SAMPLECOLL.LOWER_COST AS
SELECT SUPPLIER_NUMBER, A.ITEM_NUMBER, UNIT_COST, SUPPLIER_COST
FROM SAMPLECOLL.INVENTORY_LIST A, SAMPLECOLL.SUPPLIERS B
WHERE A.ITEM_NUMBER = B.ITEM_NUMBER
AND UNIT_COST > SUPPLIER_COST
```

The following table is the result of running the SQL statement:

```
SELECT * FROM SAMPLECOLL.LOWER_COST
```

```

                                Display Data
                                Data width . . . . . :    51
Position to line . . . . .      Shift to column . . . . .
.....1.....2.....3.....4.....5.
SUPPLIER_NUMBER  ITEM          UNIT  SUPPLIER_COST
                NUMBER        COST
      1234        229740        1.50          1.00
      9988        153047        10.00         8.00
      2424        153047        10.00         9.00
      3366        303476         2.00         1.50
      3366        073956        20.00        17.00
***** End of data *****
                                Bottom
F3=Exit      F12=Cancel      F19=Left      F20=Right      F21=Split

```

Note: Since no ORDER BY clause was specified for the query, the order of the rows returned by your query may be different.

The rows that can be seen through this view are only those rows that have a supplier cost that is less than the unit cost.

For more information about using Interactive SQL, see Chapter 17, “Using Interactive SQL” on page 285.

Chapter 3. Getting started with iSeries Navigator Database

This chapter describes how to create and work with libraries (schemas or SQL collections), tables, and views using iSeries Navigator. iSeries Navigator Database is a graphical interface that you can use to perform many of your common administrative database operations. Most of the iSeries Navigator operations are based on Structured Query Language (SQL), but you do not need to fully understand SQL to perform them. Chapter 16, “Advanced database functions using iSeries Navigator” on page 271 explains some advanced database functions that use iSeries Navigator. In this chapter, the examples use iSeries Navigator to show the execution of common database tasks. The objects created are the same objects that are created in the examples using Interactive SQL in Chapter 2, “Getting Started with SQL” on page 15.

See the following topics for details:

- “Starting iSeries Navigator”
- “Creating a library with iSeries Navigator”
- “Edit list of libraries displayed in iSeries Navigator” on page 35
- “Creating and using a table using iSeries Navigator” on page 35
- “Defining columns on a table using iSeries Navigator” on page 36
- “Copying column definitions using iSeries Navigator” on page 38
- “Inserting information into a table using iSeries Navigator” on page 38
- “Viewing the contents of a table using iSeries Navigator” on page 40
- “Copying and moving a table using iSeries Navigator” on page 41
- “Creating and using a view with iSeries Navigator” on page 42
- “Deleting database objects using iSeries Navigator” on page 46

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Starting iSeries Navigator

To start iSeries Navigator for the following examples:

1. Double click on the **iSeries Navigator** icon
2. Expand the system you want to use.

For more information about Setting up iSeries Navigator, see Getting to know iSeries Navigator.

Creating a library with iSeries Navigator

A library is a database structure that contains your tables, views, and other object types. You can use libraries to group related objects and to find objects by name. You can also create a library as a schema, and specify a data dictionary.

A schema (SQL collection) also includes catalog views that contain descriptions and information for all tables, views, indexes, files, packages, and constraints created in the library. All tables created in the schema automatically have journaling performed on them.

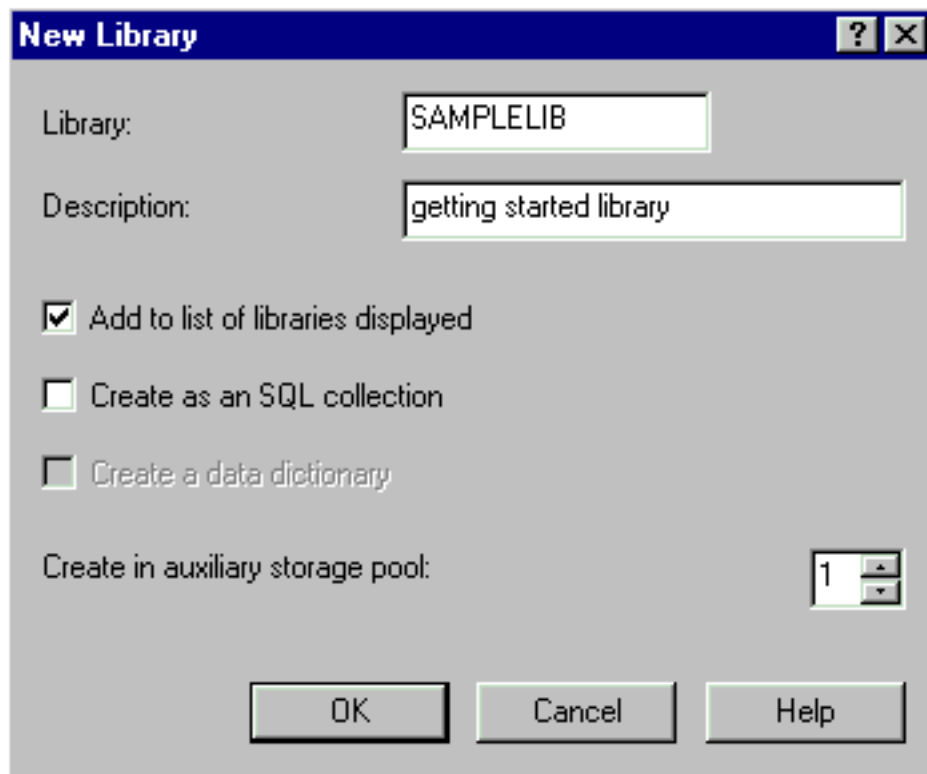
You can also work with multiple databases. See Work with multiple databases for details.

For an example of how to create a library (schema) using iSeries Navigator, see “Example: Creating a library using iSeries Navigator (SAMPLELIB)”.

Example: Creating a library using iSeries Navigator (SAMPLELIB)

You can create a sample library, named SAMPLELIB, by:

1. In the **iSeries Navigator** window, expand your server → **Database** → and the database that you want to work with.
2. Right-click **Libraries**, and select **New Library**.
3. On the **New Library** dialog, type SAMPLELIB in the name field.
4. Specify a description (optional).
5. To add to the list of libraries to be displayed, select **Add to list of libraries** displayed.
6. You can create a library as a schema by selecting **Create as a schema** and you can create a data dictionary by selecting **Create a data dictionary**. However, just create this sample library as a basic library.
7. Specify a disk pool to contain the library. Choose 1 so that the library is created on the system disk pool.
8. Click OK.



Notes:

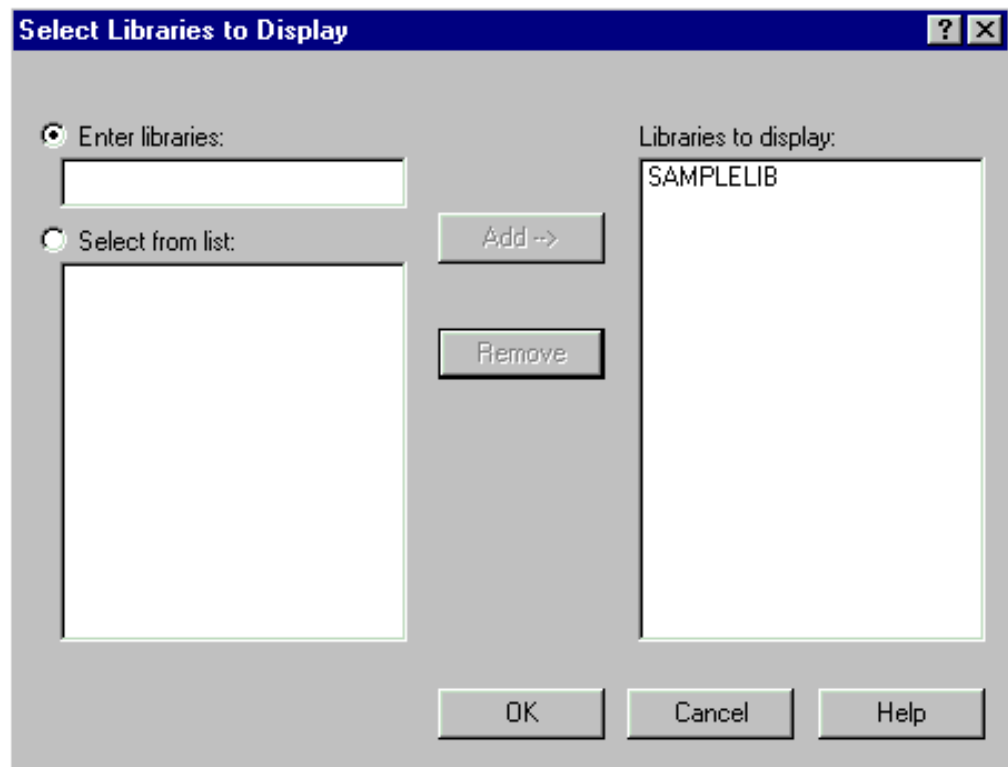
1. If we had created SAMPLELIB as a schema, several objects would be created, and this process might take several seconds.

2. See Working with multiple databases for more information about creating libraries in user disk pools.

Edit list of libraries displayed in iSeries Navigator

Once you have successfully created a library, you can create tables, views, indexes, stored procedures, user-defined function, and user-defined types in it. To edit the list of libraries displayed when you click on Libraries:

1. Right-click **Libraries**, and select **Select Libraries to Display**.
2. On the **Select Libraries to Display** dialog, you can edit the list by selecting a library name and clicking Add.
3. You can remove a library from the list of libraries to display by selecting that library from the list of libraries to display and clicking **Remove**.



Right now leave SAMPLELIB as the library displayed.

Creating and using a table using iSeries Navigator

A table is a basic database object that is used to store information. Once you have created a table, you can define the columns, create indexes, and add triggers and constraints by using the table properties dialog.

For an example of creating a table using iSeries Navigator, see “Example: Creating a table (INVENTORY_LIST) using iSeries Navigator” on page 36.

When you are creating a table, you need to understand the concepts of null value and default value. A null value indicates the absence of a column value for a row. It is not the same as a value of zero or all blanks. It means “unknown”. It is not

equal to any value, not even to other null values. If a column does not allow the null value, a value must be assigned to the column. This value is either a default value or a user supplied value.

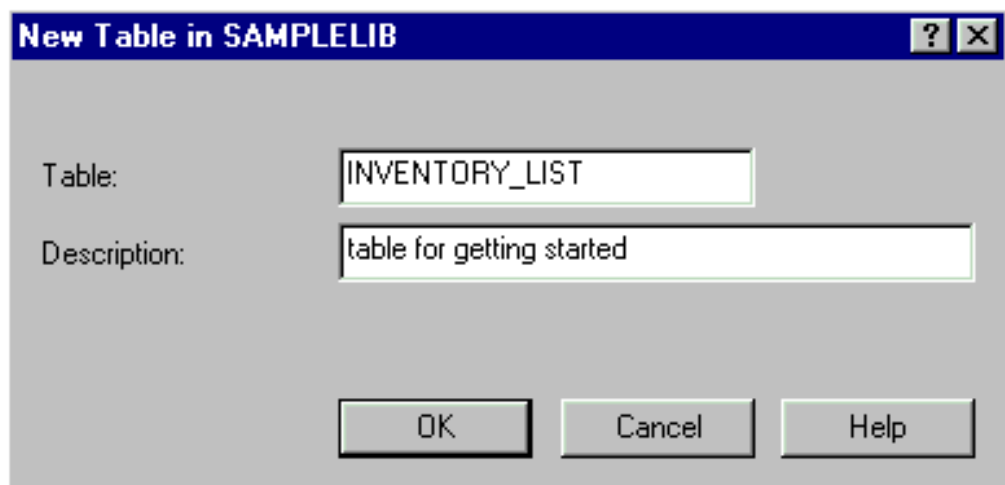
If no value is specified for a column when a row is added to a table, the row is assigned a default value. If the column is not assigned a specific default value, the column uses the system default value. For more information about the default values used by INSERT, see “Inserting rows using the INSERT statement” on page 93

Example: Creating a table (INVENTORY_LIST) using iSeries Navigator

We are going to create a table to maintain information about the current inventory of a business. It will have information about the items kept in the inventory, their cost, quantity currently on hand, the last order date, and the number last ordered. The item number will be a required value. It cannot be null. The item name, quantity on hand, and order quantity will have user-supplied default values. The last order date and quantity will allow the null value.

To create a table:

1. In the **iSeries Navigator** window, expand your server → **Database** → the database you want to work with → **Libraries**.
2. Right-click SAMPLELIB and select **New**.
3. Select **Table**.
4. On the **New Table in** dialog, type INVENTORY_LIST as the table name.
5. Specify a description (optional).
6. Click **OK**.

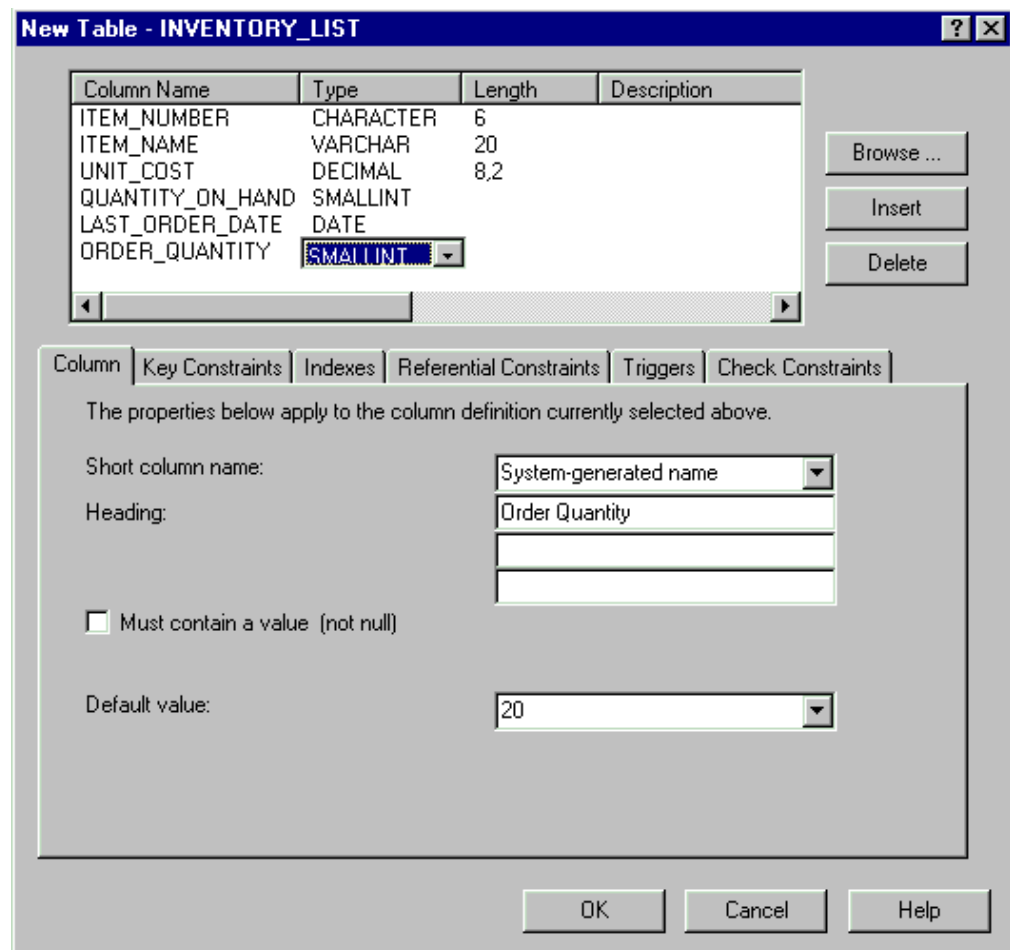


The New Table - INVENTORY_LIST appears. Do not close this dialog; we will need it for the next step.

Defining columns on a table using iSeries Navigator

You can define columns on a new or existing table. If you are defining columns for an existing table, navigate to the table by expanding **Database** → **Libraries** → **SAMPLELIB** (or appropriate library name). In the detail pane, right-click the table *INVENTORY_LIST* and select **Properties**.

1. To define a column on the **Table Properties** or **New Table** dialog, click **New** or **Insert**. A new column will appear in the **Column definition grid**.
2. In the **Column definition grid**, enter the name *ITEM_NUMBER*.
3. Make sure that the type is CHARACTER. You can change types by clicking on the type currently listed, clicking the down arrow, and selecting a new type from the list provided.
4. Specify a length of 6 for this column. For data types where the size is predetermined, the size is filled in and you cannot change the value.
5. You can specify a description for the column. This step is optional.
6. Below on the **Column** tab, you can specify a short name in the **Short column name** text box. If you do not specify a short name, the system automatically generates a name. If the column name is 10 characters or less, then the Short name is the same as the Column name. You can perform queries by using either column name. Just leave this space blank for now.
7. Enter a column heading for each column.
8. Select **Must contain a value (not null)**. This ensures that a value must be placed in this column in order for the row insert to be successful.
9. Make sure to set the default value at **No default**. This is a column that has to have a value entered.



You have now defined column ITEM_NUMBER. Add the following columns to Table INVENTORY_LIST:

Column name	Type	Length	Scale	Null	Default Value
ITEM_NAME	VARCHAR	20		Not null	UNKNOWN
UNIT_COST	DECIMAL	8	2	Not null	(set to column data type default)
QUANTITY_ON_HAND	SMALLINT			NULL	NULL
LAST_ORDER_DATE	DATE			NULL	
ORDER_QUANTITY	SMALLINT			NULL	20

When you are finished defining these columns, click **OK** to create the table.

Creating the supplier table (SUPPLIERS) using iSeries Navigator

Later in our examples, we will need a second table as well. This table will contain information about suppliers of our inventory items, which items they supply, and the cost of the item from that supplier. Create a table called SUPPLIERS in SAMPLELIB. This table will have three columns: SUPPLIER_NUMBER, ITEM_NUMBER, and SUPPLIER_COST. Notice that this table has a common column with table INVENTORY_LIST: ITEM_NUMBER. Rather than create a new ITEM_NUMBER column, we can copy the column definition used for ITEM_NUMBER in INVENTORY_LIST table.

Copying column definitions using iSeries Navigator

To copy column definitions:

1. On the **SUPPLIER Table Properties** or the **New Table** dialog, click **Browse**.
2. On the **Browse Tables** dialog, expand SAMPLELIB.
3. Click INVENTORY_LIST. The columns in that table are listed, along with their data type, size, and description.
4. Select ITEM_NUMBER.
5. Click **OK** to copy this column definition to table SUPPLIERS.

Add the last two columns for table SUPPLIERS with the following values: SUPPLIER_NUMBER, CHAR(4), NOT NULL and SUPPLIER_COST, DECIMAL (8,2)

Inserting information into a table using iSeries Navigator

To insert, edit or delete data in a table, you must have authority to that table. To add data to the table INVENTORY_LIST:

1. In the **iSeries Navigator** window, expand your server → **Database** → the database you want to work with → **Libraries**.
2. Click SAMPLELIB.
3. Right-click INVENTORY_LIST and select **Open**.
4. From the Rows menu, select **Insert**. A new row appears.
5. Enter the following information under the appropriate headings.

SAMPLELIB.INVENTORY_LIST						
ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND	LAST_ORDER_DATE	ORDER_QUANTITY	
153047	Pencils, red	10.00	25		20	
229740	Lined tablets	1.50	120		20	
544931	UNKNOWN	5.00			20	
303476	Paper clips	2.00	100		20	
559343	Envelopes, legal	3.00	500		20	
291124	Envelopes, stand...	0			20	
775298	Chairs, secretary	225.00	6		20	
073956	Pens, black	20.00	25		20	

Note: The values you enter must satisfy all constraints and satisfy the type of each column. If there is a unique constraint or index over the table, the values you enter must define a unique key value. If you do not enter a value in a column, the default value will be entered, if allowed. For this exercise, do not insert values that are not shown in the chart below so that the default values are used.

ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND
153047	Pencils, red	10.00	25
229740	Lined tablets	1.50	120
544931		5.00	
303476	Paper clips	2.00	100
559343	Envelopes, legal	3.00	500
291124	Envelopes, standard		
775298	Chairs, secretary	225.00	6
073956	Pens, black	20.00	25

From the **File** menu, select **Save**.

Add the following rows to the SAMPLELIB.SUPPLIERS table.

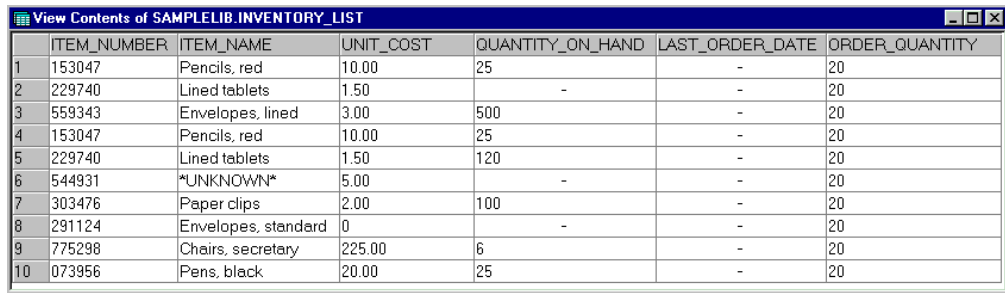
SUPPLIER_NUMBER	ITEM_NUMBER	SUPPLIER_COST
1234	153047	10.00
1234	229740	1.00
1234	303476	3.00
9988	153047	8.00
9988	559343	3.00
2424	153047	9.00
2424	303476	2.50
5546	775298	225.00
3366	303476	1.50
3366	073956	17.00

From the **File** menu, select **Save**. The sample library now contains two tables with several rows of data in each.

Viewing the contents of a table using iSeries Navigator

You can display the contents of your tables and views by using Quick View. You can only view the contents; to make changes to a table, you will have to open the table. To view the contents of INVENTORY_LIST:

1. In the **iSeries Navigator** window, expand your server → **Database** → the database that you want to work with → **Libraries**.
2. Click SAMPLELIB.
3. Right-click INVENTORY_LIST and select **Quick View**.



	ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND	LAST_ORDER_DATE	ORDER_QUANTITY
1	153047	Pencils, red	10.00	25	-	20
2	229740	Lined tablets	1.50	-	-	20
3	559343	Envelopes, lined	3.00	500	-	20
4	153047	Pencils, red	10.00	25	-	20
5	229740	Lined tablets	1.50	120	-	20
6	544931	*UNKNOWN*	5.00	-	-	20
7	303476	Paper clips	2.00	100	-	20
8	291124	Envelopes, standard	0	-	-	20
9	775298	Chairs, secretary	225.00	6	-	20
10	073956	Pens, black	20.00	25	-	20

NOTE: Quick View also allows you to access a Datalink column and its associated uniform resource locator (URL)'s, and then launches a browser when selected.

Changing information in a table using iSeries Navigator

You can use iSeries Navigator to change the data values in the columns of a table. Suppose we want to update a column using iSeries Navigator to indicate that we received an order for more paper clips today. Keep in mind that the value you enter must be valid for that column.

1. Open table INVENTORY_LIST by double-clicking on it
2. Enter the current date in the LAST_ORDER_DATE column for the row Paper clips. Be sure to use the correct date format for your system
3. Change the ORDER_QUANTITY to 50.
4. Save the changes and view the table contents by using **Quick View**.

The paper clip row reflects the changes you made.

Deleting information from a table using iSeries Navigator

You can delete data from a table by using iSeries Navigator. You can delete information from a single column in a row or delete the row entirely. Keep in mind that if a column requires a value, you will not be able to delete it without deleting the entire row.

1. Open table INVENTORY_LIST by double-clicking on it.
2. Delete the column value for ORDER_QUANTITY for the Envelopes, standard row. Because this is a column that allows Null values, we can delete the value.
3. Delete the column value for UNIT_COST for the Lined tablets row. Because this column does not allow Null values, the deletion is not allowed.

You can also delete an entire row without removing all of the column values one at a time.

1. Open table INVENTORY_LIST by double-clicking on it.
2. Click the gray cell to the left of the *UNKNOWN* row. This highlights the entire row.

3. Select **Delete** from the Rows menu or else press the Delete key on your keyboard. The *UNKNOWN* row is deleted.
4. Delete all of the rows from table INVENTORY_LIST that do not have a value in the QUANTITY_ON_HAND column.
5. Save the changes and view the contents by using Quick View. You should have a table that contains the following data:

ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND	LAST_ORDER_DATE	ORDER_QUANTITY
153047	Pencils, red	10.00	25		20
229740	Lined tablets	1.50	120		20
303476	Paper clips	2.00	100	2000-10-02	50
559343	Envelopes, legal	3.00	500		20
775298	Chairs, secretary	225.00	6		20
073956	Pens, black	20.00	25		20

Copying and moving a table using iSeries Navigator

iSeries Navigator allows you to copy or move tables from one library or system to another. Copying a table creates more than one instance of the table; moving transfers the table to its new location while removing the instance from its former location.

Create a new library called LIBRARY1 and add it to the list of libraries displayed. Once you have created this new library, copy INVENTORY_LIST over to LIBRARY1. To copy a table:

1. In the **iSeries Navigator** window, expand your server → **Database** → the database that you want to work with → **Libraries**.
2. Click on SAMPLELIB.
3. Right-click INVENTORY_LIST and select **Copy**.
4. Right-click LIBRARY and select **Paste**.

Now that you have copied table INVENTORY_LIST to LIBRARY1, move table SUPPLIERS to LIBRARY1. To move a table:

1. In the **iSeries Navigator** window, expand your server → **Database** → the database that you want to work with → **Libraries**.
2. Click on SAMPLELIB.
3. Right-click SUPPLIERS and select **Cut**.
4. Right-click LIBRARY1 and select **Paste**.

Note: You can move a table by dragging and dropping the table on the new library. Moving a table to a new location does not always remove it from the source system. For example, if you have read authority but not delete authority to the source table, you can move the table to the target system. However, you cannot delete the table from the source system, causing two instances of the table to exist.

Creating and using a view with iSeries Navigator

You may find that no single table contains all the information you need. You may also want to give users access to only part of the data in a table. Views provide a way to subset the table so that you deal with only the data you need. A view reduces complexity and, at the same time, restricts access.

In order to create a view you must have the proper authority to the tables or physical files on which the view is based. See the CREATE VIEW statement in the SQL Reference for a list of authorities needed.

If you do not specify column names in the view definition, the column names will be the same as those for the table on which the view is based.

You can make changes to a table through a view even if the view has a different number of columns or rows than the table. For INSERT, columns in the table that are not in the view must have a default value.

You can use the view as though it were a table, even though the view is totally dependent on one or more tables for data. The view has no data of its own and therefore requires no storage for the data. Because a view is derived from a table that exists in storage, when you update the view data, you are really updating data in the table. Therefore, views are automatically kept up-to-date as the tables they depend on are updated.

See “Creating and using views” on page 55 for additional information.

For examples on creating views using iSeries Navigator, see the following examples:

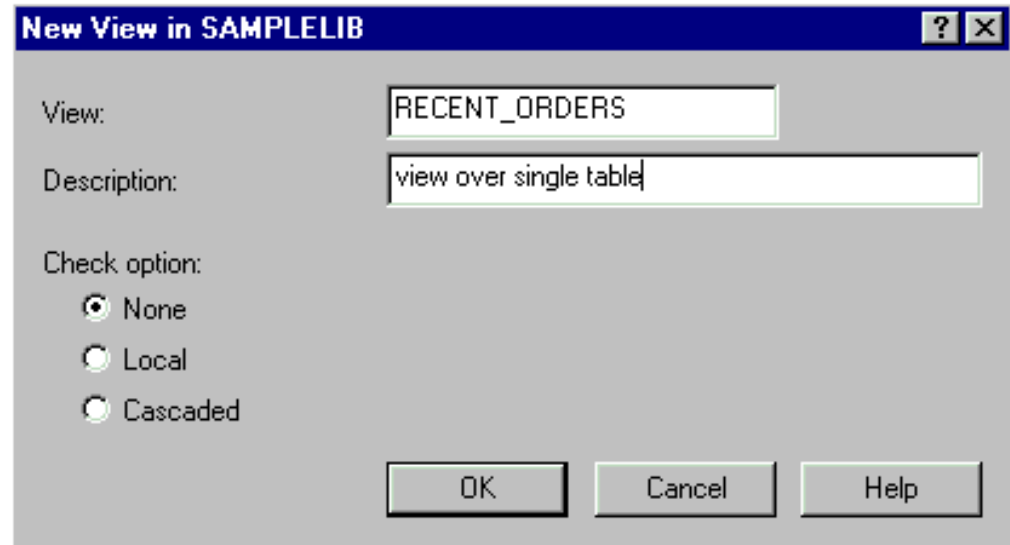
- “Creating a view over a single table using iSeries Navigator”
- “Creating a view combining data from more than one table using iSeries Navigator” on page 44

Creating a view over a single table using iSeries Navigator

The following example shows how to create a view on a single table. The view is built on the INVENTORY_LIST table. The table has six columns, but the view uses only three of the columns: ITEM_NUMBER, LAST_ORDER_DATE, and QUANTITY_ON_HAND.

To create a view over a single table:

1. In the **iSeries Navigator** window, expand your server → **Database** → the database that you want to work with → **Libraries**.
2. Right-click on SAMPLELIB and select **New**, then **View**.
3. On the **New view** dialog, type RECENT_ORDERS in the **Name** field.
4. Optionally, you can specify a description.
5. Additionally, on this dialog, select a check option. A check option on a view specifies that the values inserted or updated into a row must conform to the conditions of the view. For more information about check option, see “WITH CHECK OPTION on a View” on page 144. For this view, select **None**.
6. Click **OK**.

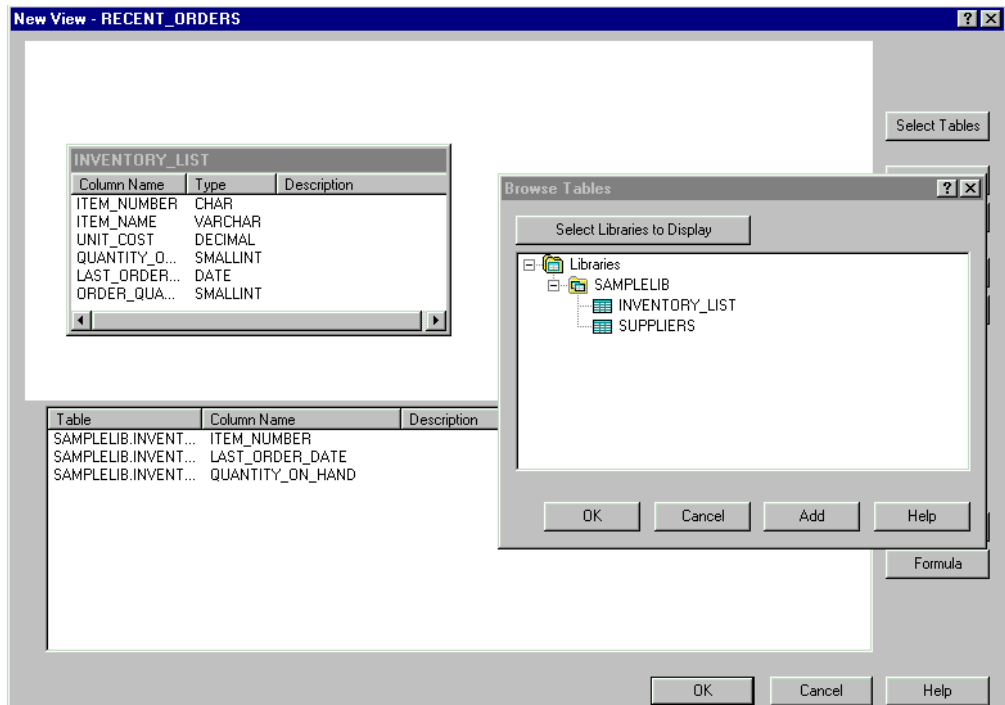


On the **New view** dialog:

1. Click **Select tables**.
2. On the **Browse for tables** dialog, expand SAMPLELIB, then select INVENTORY_LIST.
3. Click **Add**.
4. Click **OK**. INVENTORY_LIST should now be in the work area on the New View dialog.
5. To choose the columns that you want in the new view, click on them in the selected tables and drag and drop them in the selection grid on the bottom half of the dialog. Select ITEM_NUMBER, LAST_ORDER_DATE, and QUANTITY_ON_HAND.
6. The order that the columns appear in the selection grid is the order that they will appear in the view. To change the order, select a column and drag it to its new position. Put the columns in the following order: ITEM_NUMBER LAST_ORDER_DATE QUANTITY_ON_HAND.

The view is now essentially finished, but we only want to view those items that have been ordered in the last 14 days. To specify this information, we need to create a WHERE clause:

1. Click **Select Rows**.
2. On the **Select Rows** dialog, type the following: WHERE LAST_ORDER_DATE > CURRENT DATE - 14 DAYS. You can select the elements that make up this WHERE clause by selecting them from the options shown.
3. Click **OK**.
4. To view the SQL used to generate this view, click **Show SQL**.
5. Click **OK** to create the view.



To display the contents of RECENT_ORDERS, right-click on RECENT_ORDERS and select **QUICK VIEW**. You should see the following information displayed:

ITEM_NUMBER	LAST_ORDER_DATE	QUANTITY_ON_HAND
303476	2000-10-02	100

In the example above, the columns in the view have the same name as the columns in the table because we did not specify new names. The schema that the view is created into does not need to be the same schema as the table it is built over. You could use any schema or library.

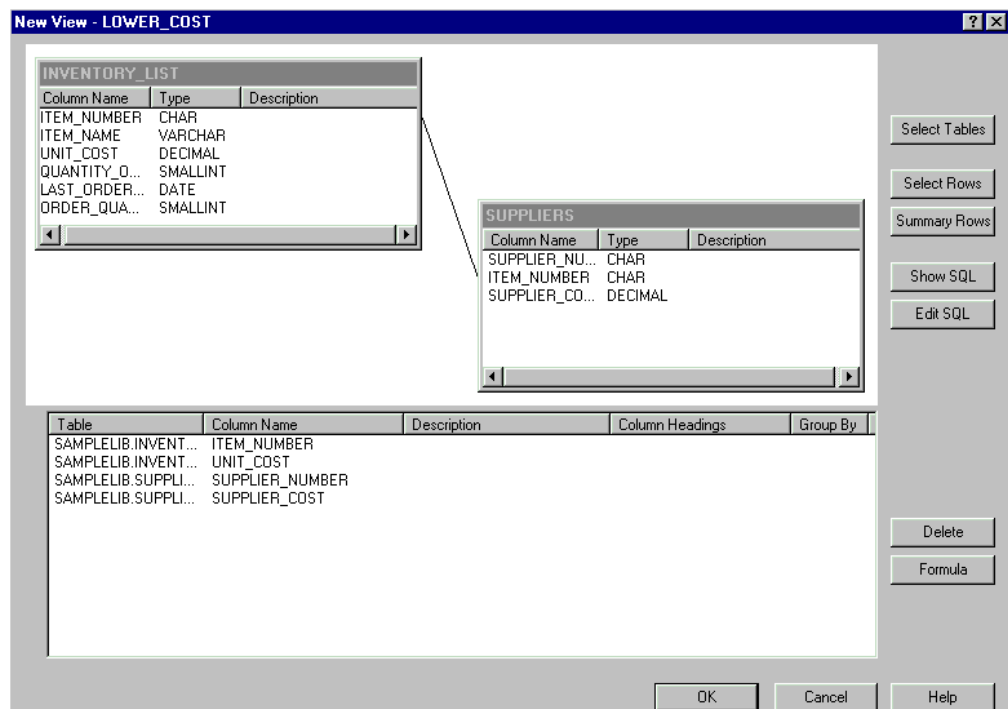
Creating a view combining data from more than one table using iSeries Navigator

You can create a view combining information from more than one table by selecting more than one table in the work area of the New View dialog. You can create a simple view from more than one table by selecting the columns that you want to include from different tables and clicking OK. However, this example shows how to create a view that joins information from two different tables and returns only those rows that we want to see, much like using a WHERE clause.

Create a view that contains only those item numbers for suppliers that can supply an item at lower cost than the current unit cost. This will require selecting ITEM_NUMBER and UNIT_COST from the INVENTORY_LIST table and joining them with SUPPLIER_NUMBER and SUPPLIER_COST from the SUPPLIERS table. A WHERE clause is used to limit the number of rows returned.

1. Create a view called LOWER_COST.
2. On the **New VIEW** dialog, click **Select Tables**.
3. Select INVENTORY_LIST from SAMPLELIB and SUPPLIERS from LIBRARY1.
4. Click **OK**. Both tables should appear in the working area of the dialog.

5. Select ITEM_NUMBER and UNIT_COST from INVENTORY_LIST.
6. Select SUPPLIER_NUMBER and SUPPLIER_COST from SUPPLIERS.
7. To define the join, select ITEM_NUMBER from INVENTORY_LIST and drag it to ITEM_NUMBER in SUPPLIERS. A line is drawn from one column to the other and the **Join** dialog opens.
8. On the **Join** dialog, select **Inner Join**. For more information about Joins, see “Joining data from more than one table” on page 78.
9. Click **OK**.
10. Once again, you can view the SQL used to create this view by selecting **Show SQL**. You can also edit the SQL by selecting **Edit SQL**. **Edit SQL** launches **Run SQL Scripts** where you can edit your SQL statement. Be aware, though, that if you change the SQL, you will need to run the statement from **Run SQL Scripts** rather than returning to the **New View** dialog. If you return to the **New View** dialog, your changes are not saved.
11. Click on **Select Rows** to create a WHERE clause for the view. Double-click on SUPPLIER_COST, then double-click on the < operator and finally double-click on UNIT_COST. As you click on the items, they appear in the dialog. You could also have typed this in directly.
12. Click **OK** to create the view, LOWER_COST.



To display the contents of this new view, right-click LOWER_COST and select **Quick View**. The rows that you see through this view are only those rows that have a supplier cost that is less than the unit cost.

SUPPLIER_NUMBER	ITEM_NUMBER	UNIT_COST	SUPPLIER_COST
9988	153047	10.00	8.00
2424	153047	10.00	9.00
1234	229740	1.50	1.00
3366	303476	2.00	1.50

SUPPLIER_NUMBER	ITEM_NUMBER	UNIT_COST	SUPPLIER_COST
3366	073956	20.00	17.00

Deleting database objects using iSeries Navigator

Once you have created these objects on your system, you may want to drop them to save on system resource. You will need Delete authority in order to perform these tasks.

NOTE: If you would like to retain the information in these tables, create a third library and copy the tables and views to it. First, drop INVENTORY_LIST table from LIBRARY1:

1. In the **iSeries Navigator** window, expand your server → **Database** → the database that you want to work with → **Libraries**.
2. Expand LIBRARY1.
3. Right-click INVENTORY_LIST and select **Delete** or else hit the Delete key.
4. On the **Object deletion confirmation** dialog, select **Delete**. INVENTORY_LIST table is dropped.

Next, delete SUPPLIERS from LIBRARY1:

1. Right-click SUPPLIERS and select **Delete** or else hit the Delete key.
2. On the object deletion confirmation dialog, select **Yes**.
3. A new dialog opens, indicating that the view, LOWER_COST is dependent on SUPPLIERS and if it too, should be deleted. Click **Delete**.

SUPPLIERS and LOWER_COST are deleted. Now that LIBRARY1 is empty, delete it by right-clicking on it and selecting Delete. On the object deletion confirmation dialog, select Yes. LIBRARY1 is deleted.

Finally, delete SAMPLELIB:

1. In the **iSeries Navigator** window, expand your server → **Database** → the database that you want to work with → **Libraries**.
2. Right-click SAMPLELIB and select **Delete**.
3. On the object deletion confirmation dialog, select **Delete**.
4. A new dialog opens, indicating that the table INVENTORY_LIST and view RECENT_ORDERS are dependent on INVENTORY_LIST and if they, too, should be deleted. Click **Yes**.

SAMPLELIB, INVENTORY_LIST, and RECENT_ORDERS are deleted.

For more information about using iSeries Navigator, see Chapter 16, “Advanced database functions using iSeries Navigator” on page 271.

Chapter 4. Data Definition Language (DDL)

Data Definition Language (DDL) describes the portion of SQL that allows you to create, alter, and destroy database objects. These database objects include schemas, tables, views, catalogs, indexes, and aliases.

For details, see the following sections:

- “Creating a schema”
- “Creating a table” on page 48
- “Creating a table using LIKE” on page 48
- “Creating a table using AS” on page 49
- “Declaring a global temporary table” on page 49
- “Creating and altering an identity column” on page 50
- “ROWID” on page 51
- “Creating descriptive labels using the LABEL ON statement” on page 51
- “Describing an SQL object using COMMENT ON” on page 52
- “Changing a table definition” on page 52
- “Creating and using ALIAS names” on page 55
- “Creating and using views” on page 55
- “Adding indexes” on page 57
- “Catalogs in database design” on page 58
- “Dropping a database object” on page 59

Creating a schema

A schema provides a logical grouping of SQL objects. A schema consists of a library, a journal, a journal receiver, a catalog, and optionally, a data dictionary. Tables, views, and system objects (such as programs) can be created, moved, or restored into any system library. All system files can be created or moved into an SQL schema if the SQL schema does not contain a data dictionary. If the SQL schema contains a data dictionary then:

- Source physical files or nonsource physical files with one member can be created, moved, or restored into an SQL schema.
- Logical files cannot be placed in an SQL schema because they cannot be described in the data dictionary.

You can create and own many schemas.

Schemas are created using the CREATE SCHEMA statement. For example:

Create a schema called DBTEMP.

```
CREATE SCHEMA DBTEMP
```

You can use the term *collection* synonymously with *schema*. For more information on the CREATE SCHEMA statement, see CREATE SCHEMA in the *SQL Reference* book.

Creating a table

A table can be visualized as a two-dimensional arrangement of data consisting of rows and columns. The row is the horizontal part containing one or more columns. The column is the vertical part containing one or more rows of data of one data type. All data for a column must be of the same type. A table in SQL is a keyed or nonkeyed physical file. See the Data types topic in the SQL Reference book for a description of data types.

Tables are created using the CREATE TABLE statement. The definition must include its name and the names and attributes of its columns. The definition may include other attributes of the table such as primary key.

Example 1: Given that you have administrative authority, create a table named 'INVENTORY' with the following columns:

- Part number: Integer between 1 and 9 999, must not be null
- Description: Character of length 0 to 24
- Quantity on hand: Integer between 0 and 100000

The primary key is PARTNO.

```
CREATE TABLE INVENTORY
              (PARTNO      SMALLINT      NOT NULL,
              DESCR       VARCHAR(24 ),
              QONHAND     INT,
              PRIMARY KEY(PARTNO))
```

Adding constraints to a table

Constraints can be added to a new table or an existing table. You can add a unique or primary key, a referential constraint, or a check constraint, using the *ADD constraint* clause on the CREATE TABLE or the ALTER TABLE statements. For example, add a primary key to a new table or to an existing table. The following example illustrates adding a primary key to an existing table using the ALTER TABLE statement.

```
ALTER TABLE CORPDATA.DEPARTMENT
  ADD PRIMARY KEY (DEPTNO)
```

To make this key a unique key, simply substitute the keyword PRIMARY with UNIQUE. See Chapter 10, "Data Integrity" on page 135 for more details.

Creating a table using LIKE

You can create a table that looks like another table. That is, you can create a table that includes all of the column definitions from an existing table. The definitions that are copied are:

- Column names (and system column names)
- Data type, precision, length, and scale
- CCSID
- Column text (LABEL ON)
- Column heading (LABEL ON)

If the LIKE clause immediately follows the table name and is not enclosed in parenthesis, the following attributes are also included:

- Default value
- Nullability

If the specified table or view contains an identity column, you must specify `INCLUDING IDENTITY` on the `CREATE TABLE` statement if you want the identity column to exist in the new table. The default behavior for `CREATE TABLE` is `EXCLUDING IDENTITY`. If the specified table or view is a non-SQL created physical file or logical file, any non-SQL attributes are removed.

Create a table `EMPLOYEE2` that includes all of the columns in `EMPLOYEE`.

```
CREATE TABLE EMPLOYEE2 LIKE EMPLOYEE
```

For complete details about `CREATE TABLE LIKE`, see `CREATE TABLE` in the *SQL Reference* topic.

Creating a table using AS

`CREATE TABLE AS` creates a table from the result of a `SELECT` statement. All of the expressions that can be used in a `SELECT` statement can be used in a `CREATE TABLE AS` statement. You can also include all of the data from the table or tables that you are selecting from.

For example, create a table named `EMPLOYEE3` that includes all of the column definitions from `EMPLOYEE` where the `DEPTNO = D11`.

```
CREATE TABLE EMPLOYEE3 AS  
  (SELECT PROJNO, PROJNAME, DEPTNO  
   FROM EMPLOYEE  
   WHERE DEPTNO = 'D11') WITH NO DATA
```

If the specified table or view contains an identity column, you must specify `INCLUDING IDENTITY` on the `CREATE TABLE` statement if you want the identity column to exist in the new table. The default behavior for `CREATE TABLE` is `EXCLUDING IDENTITY`. The `WITH NO DATA` clause says that the column definitions are to be copied without the data. If you wanted to include the data in the new table, `EMPLOYEE3`, you would say `WITH DATA`. For more information about using `SELECT`, see Chapter 5, “Retrieving data using the `SELECT` statement” on page 61. If the specified query includes a non-SQL created physical file or logical file, any non-SQL result attributes are removed. For complete details about `CREATE TABLE AS`, see `CREATE TABLE` in the *SQL Reference* topic.

Declaring a global temporary table

You can create a temporary table for use with your current session using the `DECLARE GLOBAL TEMPORARY TABLE` statement. This temporary table does not appear in the system catalog and cannot be shared by other sessions. When you terminate your session, the rows of the table are deleted and the table is dropped.

The syntax of this statement is similar to `CREATE TABLE`, including the `LIKE` and `AS` clause.

For example, create a temporary table `ORDERS`:

```
DECLARE GLOBAL TEMPORARY TABLE ORDERS  
  (PARTNO SMALLINT NOT NULL,  
   DESCR VARCHAR(24),  
   QONHAND INT)  
ON COMMIT DELETE ROWS
```

This table is created in QTEMP. To reference the table using a schema name, use either SESSION or QTEMP. You can issue SELECT, INSERT, UPDATE, and DELETE statements against this table just as you would any other table. You can drop this table by issuing the DROP TABLE statement:

```
DROP TABLE ORDERS
```

For complete details, see DECLARE GLOBAL TEMPORARY TABLE in the *SQL Reference* topic.

Creating and altering an identity column

Every time that a new row is added to a table with an identity column, the identity column value in the new row is incremented (or decremented) by the system. Only columns of type SMALLINT, INTEGER, BIGINT, DECIMAL, or NUMERIC can be created as identity columns. You are allowed only one identity column per table. When you are changing a table definition, only a column that you are adding can be specified as an identity column; existing columns cannot.

When you create a table, you can define a column in the table to be an identity column. For example, create a table ORDERS with 3 columns called ORDERNO, SHIPPED_TO, and ORDER_DATE. Define ORDERNO as an identity column.

```
CREATE TABLE ORDERS  
  (ORDERNO SMALLINT NOT NULL  
   GENERATED ALWAYS AS IDENTITY  
   (START WITH 500  
   INCREMENT BY 1  
   CYCLE) ,  
  SHIPPED_TO VARCHAR (36) ,  
  ORDER_DATE DATE)
```

This column is defined with starting value of 500, incremented by 1 for every new row inserted, and will recycle when the maximum value is reached. In this example, the maximum value for the identity column is the maximum value for the data type. Because the data type is defined as SMALLINT, the range of values that can be assigned to ORDERNO is from 500 to 32767. When this column value reaches 32767, it will restart at 500 again. If 500 is still assigned to a column, and a unique key is specified on the identity column, then a duplicate key error is returned. The next insert will attempt to use 501. If you do not have a unique key specified for the identity column, 500 is used again, regardless of how many times it appears in the table.

For a larger range of values, you could specify the column to be an INTEGER or even a BIGINT. If you wanted the value of the identity column to decrease, you would specify a negative value for the INCREMENT option. It is also possible to specify the exact range of numbers by using MINVALUE and MAXVALUE.

You can modify the attributes of an existing identity column using the ALTER TABLE statement. For example, if you wanted to restart the identity column with a new value:

```
ALTER TABLE ORDER  
  ALTER COLUMN ORDERNO  
  RESTART WITH 1
```

You can also drop the identity attribute from a column:

```
ALTER TABLE ORDER  
  ALTER COLUMN ORDERNO  
  DROP IDENTITY
```


The column ORDERNO remains as a SMALLINT column, but the identity attribute is dropped. The system will no longer generate values for this column.

ROWID

Using ROWID is another way to have the system assign a unique value to a column in a table. ROWID is similar to identity columns, but rather than being an attribute of a numeric column, it is a separate data type. To create a table similar to the identity column example:

```
CREATE TABLE ORDERS
  (ORDERNO ROWID
   GENERATED ALWAYS,
   SHIPPED_TO VARCHAR (36) ,
   ORDER_DATE DATE)
```

Creating descriptive labels using the LABEL ON statement

Sometimes the table name, column name, view name, alias name, or SQL package name does not clearly define data that is shown on an interactive display of the table. By using the LABEL ON statement, you can create a more descriptive label for the table name, column name, view name, alias name, or SQL package name. These labels can be seen in the SQL catalog in the LABEL column.

The LABEL ON statement looks like this:

```
LABEL ON
  TABLE CORPDATA.DEPARTMENT IS 'Department Structure Table'

LABEL ON
  COLUMN CORPDATA.DEPARTMENT.ADMRDEPT IS 'Reports to Dept.'
```

After these statements are run, the table named DEPARTMENT displays the text description as *Department Structure Table* and the column named ADMRDEPT displays the heading *Reports to Dept*. The label for tables, views, SQL packages, and column text cannot be more than 50 characters and the label for column headings cannot be more than 60 characters (blanks included). The following are examples of LABEL ON statements for column headings:

This LABEL ON statement provides column heading 1 and column heading 2.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.EMPNO IS
  'Employee          Number'
```

This LABEL ON statement provides 3 levels of column headings for the SALARY column.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS
  'Yearly          Salary          (in dollars)'
```

This LABEL ON statement removes the column heading for SALARY.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS ''
```

An example of a DBCS column heading with two levels specified.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS
  '<AABCCDD>          <EEFFGG>'
```

This LABEL ON statement provides column text for the EDLEVEL column.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.EDLEVEL TEXT IS
    'Number of years of formal education'
```

For more information about the LABEL ON statement, see the LABEL ON statement in the SQL Reference book.

Describing an SQL object using COMMENT ON

After you create an SQL object such as a table, view, index, package, procedure, parameter, user-defined type, function, or trigger, you can supply information about it for future referral, such as the purpose of the object, who uses it, and anything unusual or special about it. You can also include similar information about each column of a table or view. Your comment must not be more than 2000 bytes. For more information about the COMMENT ON statement, see COMMENT ON in the *SQL Reference* book.

A comment is especially useful if your names do not clearly indicate the contents of the columns or objects. In that case, use a comment to describe the specific contents of the column or objects.

An example of using COMMENT ON follows:

```
COMMENT ON TABLE CORPDATA.EMPLOYEE IS
    'Employee table. Each row in this table represents
    one employee of the company.'
```

Getting comments after running a COMMENT ON statement

After running a COMMENT ON statement for a table, your comments are stored in the *LONG_COMMENT* column of SYSTABLES. Comments for the other objects are stored in the *LONG_COMMENT* column of the appropriate catalog table. (If the indicated row had already contained a comment, the old comment is replaced by the new one.) The following example gets the comments added by the COMMENT ON statement in the previous example:

```
SELECT LONG_COMMENT
FROM CORPDATA.SYSTABLES
WHERE NAME = 'EMPLOYEE'
```

Changing a table definition

Changing the definition of a table allows you to add new columns, change an existing column definition (change its length, default value, and so on), drop existing columns, and add and remove constraints. Table definitions are changed using the SQL ALTER TABLE statement.

You can add, change, or drop columns and add or remove constraints all with one ALTER TABLE statement. However, a single column can be referenced only once in the ADD COLUMN, ALTER COLUMN, and DROP COLUMN clauses. That is, you cannot add a column and then alter that column in the same ALTER TABLE statement.

For more information, see the following topics:

- “Adding a column” on page 53
- “Changing a column” on page 53
- “Allowable conversions” on page 53
- “Deleting a column” on page 54

- “Order of operations for ALTER TABLE statement” on page 54

Adding a column

You can add a column to a table using the ADD COLUMN clause of the SQL ALTER TABLE statement.

When you add a new column to a table, the column is initialized with its default value for all existing rows. If NOT NULL is specified, a default value must also be specified.

The altered table may consist of up to 8000 columns. The sum of the byte counts of the columns must not be greater than 32766 or, if a VARCHAR or VARGRAPHIC column is specified, 32740. If a LOB column is specified, the sum of record data byte counts of the columns must not be greater than 15 728 640.

Changing a column

You can change a column definition in a table using the ALTER COLUMN clause of the ALTER TABLE statement. When you change the data type of an existing column, the old and new attributes must be compatible. “Allowable conversions” shows the conversions with compatible attributes.

When you convert to a data type with a longer length, data will be padded with the appropriate pad character. When you convert to a data type with a shorter length, data may be lost due to truncation. An inquiry message prompts you to confirm the request.

If you have a column that does not allow the null value and you want to change it to now allow the null value, use the DROP NOT NULL clause. If you have a column that allows the null value and you want to prevent the use of null values, use the SET NOT NULL clause. If any of the existing values in that column are the null value, the ALTER TABLE will not be performed and an SQLCODE of -190 will result.

Allowable conversions

Table 2. Allowable Conversions

FROM data type	TO data type
Decimal	Numeric
Decimal	Bigint, Integer, Smallint
Decimal	Float
Numeric	Decimal
Numeric	Bigint, Integer, Smallint
Numeric	Float
Bigint, Integer, Smallint	Decimal
Bigint, Integer, Smallint	Numeric
Bigint, Integer, Smallint	Float
Float	Numeric
Float	Bigint, Integer, Smallint
Character	DBCS-open
Character	UCS-2 graphic

Table 2. Allowable Conversions (continued)

FROM data type	TO data type
DBCS-open	Character
DBCS-open	UCS-2 graphic
DBCS-either	Character
DBCS-either	DBCS-open
DBCS-either	UCS-2 graphic
DBCS-only	DBCS-open
DBCS-only	DBCS graphic
DBCS-only	UCS-2 graphic
DBCS graphic	UCS-2 graphic
UCS-2 graphic	Character
UCS-2 graphic	DBCS-open
UCS-2 graphic	DBCS graphic
distinct type	source type
source type	distinct type

When modifying an existing column, only the attributes that you specify will be changed. All other attributes will remain unchanged. For example, given the following table definition:

```
CREATE TABLE EX1 (COL1 CHAR(10) DEFAULT 'COL1',
                  COL2 VARCHAR(20) ALLOCATE(10) CCSID 937,
                  COL3 VARGRAPHIC(20) ALLOCATE(10)
                  NOT NULL WITH DEFAULT)
```

After running the following ALTER TABLE statement:

```
ALTER TABLE EX1 ALTER COLUMN COL2 SET DATA TYPE VARCHAR(30)
ALTER COLUMN COL3 DROP NOT NULL
```

COL2 would still have an allocated length of 10 and CCSID 937, and COL3 would still have an allocated length of 10.

Deleting a column

You can delete a column using the DROP COLUMN clause of the ALTER TABLE statement.

Dropping a column deletes that column from the table definition. If CASCADE is specified, any views, indexes, and constraints dependent on that column will also be dropped. If RESTRICT is specified, and any views, indexes, or constraints are dependent on the column, the column will not be dropped and SQLCODE of -196 will be issued.

```
ALTER TABLE DEPT
DROP COLUMN NUMDEPT
```

Order of operations for ALTER TABLE statement

An ALTER TABLE statement is performed as a set of steps as follows:

1. Drop constraints
2. Drop columns for which the RESTRICT option was specified

3. Alter column definitions (this includes adding columns and dropping columns for which the CASCADE option was specified)
4. Add constraints

Within each of these steps, the order in which you specify the clauses is the order in which they are performed, with one exception. If any columns are being dropped, that operation is logically done before any column definitions are added or altered, in case record length is increased as a result of the ALTER TABLE statement.

Creating and using ALIAS names

When you refer to an existing table or view, or to a physical file that consists of multiple members, you can avoid using file overrides by creating an alias. You can use the SQL CREATE ALIAS statement to do this.

You can create an alias for

- A table or view
- A *member* of a table

A table alias defines a name for the file, including the specific member name. You can use this alias name in an SQL statement in the same way that you would use a table name. Unlike overrides, alias names are objects that exist until they are dropped.

For example, if there is a multiple member file MYLIB.MYFILE with members MBR1 and MBR2, an alias can be created for the second member so that SQL can easily refer to it.

```
CREATE ALIAS MYLIB.MYMBR2_ALIAS FOR MYLIB.MYFILE (MBR2)
```

When alias MYLIB.MYMBR2_ALIAS is specified on the following insert statement, the values are inserted into member MBR2 in MYLIB.MYFILE.

```
INSERT INTO MYLIB.MYMBR2_ALIAS VALUES('ABC', 6)
```

Alias names can also be specified on DDL statements. Assume that alias MYLIB.MYALIAS exists and is an alias for table MYLIB.MYTABLE. The following DROP statement will drop table MYLIB.MYTABLE.

```
DROP TABLE MYLIB.MYALIAS
```

If you really want to drop the alias name instead, specify the ALIAS keyword on the drop statement:

```
DROP ALIAS MYLIB.MYALIAS
```

Creating and using views

A view can be used to access data in one or more tables or views. This is done by using a SELECT statement. See Chapter 5, “Retrieving data using the SELECT statement” on page 61 for detail about using the SELECT clause. For views, the ORDER BY clause cannot be used.

For example, to create a view that selects only the last name and the department of all the managers, specify:

```
CREATE VIEW CORPDATA.EMP_MANAGERS AS
  SELECT LASTNAME, WORKDEPT FROM CORPDATA.EMPLOYEE
  WHERE JOB = 'MANAGER'
```

If the select list contains elements other than columns such as expressions, functions, constants, or special registers, and the AS clause was not used to name the columns, a column list must be specified for the view. In the following example, the columns of the view are LASTNAME and YEARSOFSERVICE.

```
CREATE VIEW CORPDATA.EMP_YEARSOFSERVICE  
  (LASTNAME, YEARSOFSERVICE) AS  
  SELECT LASTNAME, YEARS (CURRENT DATE - HIREDATE)  
  FROM CORPDATA.EMPLOYEE
```

The previous view can also be defined by using the AS clause in the select list to name the columns in the view. For example:

```
CREATE VIEW CORPDATA.EMP_YEARSOFSERVICE AS  
  SELECT LASTNAME,  
         YEARS (CURRENT DATE - HIREDATE) AS YEARSOFSERVICE  
  FROM CORPDATA.EMPLOYEE
```

Once you have created the view, you can use it in SQL statements just like a table name. You can also change the data in the base table.

The following restrictions must be considered when creating the view:

- You cannot change, insert, or delete data using a read-only view. A view is read-only if it includes any of the following:
 - The first FROM clause identifies more than one table (join).
 - The first FROM clause identifies a read-only view.
 - The first FROM clause identifies a user-defined table function.
 - The first SELECT clause contains any of the SQL column functions (SUM, MAX, MIN, AVG, COUNT, STDDEV, COUNT_BIG, or VAR).
 - The first SELECT clause specifies the keyword DISTINCT.
 - The outer subselect contains a GROUP BY or HAVING clause.
 - The outer subselect contains a UNION clause.
 - A subquery where the base object of the outer-most subselect and a table of a subquery are the same table

In the above cases, you can get data from the view by means of the SQL SELECT statement, but you cannot use statements such as INSERT, UPDATE, or DELETE.

- You cannot insert a row in a view if:
 - The table on which the view is based has a column that has no default value, does not allow nulls, and is not in the view.
 - The view has a column resulting from an expression, a constant, a function, or a special register and the column was specified in the INSERT column list.
 - The WITH CHECK OPTION was specified when the view was created and the row does not match the selection criteria.
- You cannot update a column of a view that results from an expression, a constant, a function, or a special register.
- You cannot use FOR UPDATE OF, FOR READ ONLY, FETCH FIRST *n* ROWS, ORDER BY, OPTIMIZE FOR *n* ROWS, or isolation clause in the definition of a view.

Views are created with the sort sequence in effect at the time the CREATE VIEW statement is run. The sort sequence applies to all character and UCS-2 graphic comparisons in the CREATE VIEW statement subselect. See Chapter 8, "Sort sequences in SQL" on page 115 for more information about sort sequences.

Views can be created using the UNION operator. See “Creating a view with UNION” for more information.

Views can also be created using the WITH CHECK OPTION to specify the level of checking that should be done when data is inserted or updated through the view. See “WITH CHECK OPTION on a View” on page 144 for more information.

Creating a view with UNION

Using the UNION keyword, you can combine two or more subselects to form a single view. For example:

```
CREATE VIEW D11_EMPS_PROJECTS AS
  (SELECT EMPNO
   FROM CORPDATA.EMPLOYEE
   WHERE WORKDEPT = 'D11'
  UNION
  SELECT EMPNO
   FROM CORPDATA.EMPPROJECT
   WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111')
```

Results in a view with the following data:

Table 3. Creating a view as UNION results

EMPNO

000060

000150

000160

000170

000180

000190

000200

000210

000220

000230

000240

200170

200220

See “Using the UNION keyword to combine subselects” on page 85 for more detail about UNION.


Adding indexes

You can use indexes to sort and select data. In addition, indexes help the system retrieve data faster for better query performance.

Use the SQL CREATE INDEX statement to create indexes. The following example creates an index over the column *LASTNAME* in the CORPDATA.EMPLOYEE table:

```
CREATE INDEX CORPDATA.INX1 ON CORPDATA.EMPLOYEE (LASTNAME)
```

You can create any number of indexes. However, because the indexes are maintained by the system, a large number of indexes can adversely affect performance. For more information about indexes and query performance, see *Effectively Using SQL Indexes* in the *Database Performance and Query Optimization* information.

One type of index, the encoded vector index, allows for faster scans that can be more easily processed in parallel. You create encoded vector indexes by using the SQL CREATE INDEX statement. For more information about accelerating your queries with encoded vector indexes , go to the DB2 for iSeries webpages.

If an index is created that has exactly the same attributes as an existing index, the new index shares the existing indexes' binary tree. Otherwise, another binary tree is created. If the attributes of the new index are exactly the same as another index, except that the new index has fewer columns, another binary tree is still created. It is still created because the extra columns would prevent the index from being used by cursors or UPDATE statements that update those extra columns.

Indexes are created with the sort sequence in effect at the time the CREATE INDEX statement is run. The sort sequence applies to all SBCS character fields and UCS-2 graphic fields of the index. See Chapter 8, "Sort sequences in SQL" on page 115 for more information about sort sequences.

Catalogs in database design

| A catalog is automatically created when you create a schema. There is also a
| system-wide catalog that is always in the QSYS2 library. When an SQL object is
| created in a schema, information is added to both the system catalog tables and the
| schema's catalog tables. When an SQL object is created in a library, only the QSYS2
| catalog is updated. A table created with DECLARE GLOBAL TEMPORARY TABLE
| is not added to a catalog. For more information about catalogs, see the SQL
| Reference book.

As the following examples show, you can display catalog information. You cannot INSERT, DELETE, or UPDATE catalog information. You must have SELECT privileges on the catalog views to run the following examples.

Getting catalog information about a table

SYSTABLES contains a row for every table and view in the SQL schema. It tells you if the object is a table or view, the object name, the owner of the object, what SQL schema it is in, and so forth.

The following sample statement displays information for the CORPDATA.DEPARTMENT table:

```
SELECT *  
FROM CORPDATA.SYSTABLES  
WHERE NAME = 'DEPARTMENT'
```

Getting catalog information about a column

SYSCOLUMNS contains a row for each column of every table and view in the schema.

The following sample statement displays all the column names in the CORPDATA.DEPARTMENT table:


```

SELECT *
FROM CORPDATA.SYSCOLUMNS
WHERE TBNAME = 'DEPARTMENT'

```

The result of the previous sample statement is a row of information for each column in the table. Some of the information is not visible because the width of the information is wider than the display screen.

For more information about each column, specify a select-statement like this:

```

SELECT NAME, TBNAME, COLTYPE, LENGTH, DEFAULT
FROM CORPDATA.SYSCOLUMNS
WHERE TBNAME = 'DEPARTMENT'

```

In addition to the column name for each column, the select-statement shows:

- The name of the table that contains the column
- The data type of the column
- The length attribute of the column
- If the column allows default values

The result looks like this:

NAME	TBNAME	COLTYPE	LENGTH	DEFAULT
DEPTNO	DEPARTMENT	CHAR	3	N
DEPTNAME	DEPARTMENT	VARCHAR	29	N
MGRNO	DEPARTMENT	CHAR	6	Y
ADMRDEPT	DEPARTMENT	CHAR	3	N

Dropping a database object

The DROP statement deletes an object. Depending on the action requested, any objects that are directly or indirectly dependent on that object may also be deleted or may prevent the drop from happening. For example, if you drop a table, any aliases, constraints, triggers, views, or indexes associated with that table will also be dropped. Whenever an object is deleted, its description is deleted from the catalog.

For example, to drop table EMPLOYEE, issue the following statement:

```
DROP TABLE EMPLOYEE RESTRICT
```

See the DROP statement in the SQL Reference book for more details.

Chapter 5. Retrieving data using the SELECT statement

You can use the SELECT statement to retrieve data from your database. This section describes the following sections:

- “Querying data using the SELECT statement”

You can write SQL statements on one line or on many lines. For SQL statements in precompiled programs, the rules for the continuation of lines are the same as those of the host language (the language the program is written in).

Notes:

1. The SQL statements described in this section can be run on SQL tables and views, and database physical and logical files. The tables, views, and files can be either in a schema or in a library.
2. Character strings specified in an SQL statement (such as those used with WHERE or VALUES clauses) are case sensitive; that is, uppercase characters must be entered in uppercase and lowercase characters must be entered in lowercase.

WHERE ADMRDEPT='a00' (does not return a result)

WHERE ADMRDEPT='A00' (returns a valid department number)

Comparisons may not be case sensitive if a shared-weight sort sequence is being used where uppercase and lowercase characters are treated as the same character.

Querying data using the SELECT statement

You can use a variety of statements and clauses to query your data. One way to do this is to use the SELECT statement in a program to retrieve a specific row (for example, the row for an employee). Furthermore, in this example, a variety of clauses are used to gather data in a specific way. SQL provides you with several ways of tailoring your query to gather data in a specific manner. These methods are:

- “Specifying a search condition using the WHERE clause” on page 63
- “GROUP BY clause” on page 65
- “HAVING clause” on page 67
- “ORDER BY clause” on page 68

The format and syntax shown here are very basic. SELECT statements can be more varied than the examples presented in this chapter. A SELECT statement can include the following:

1. The name of each column you want to include
2. The name of the table or view that contains the data
3. A search condition to uniquely identify the row that contains the information you want
4. The name of each column used to group your data
5. A search condition that uniquely identifies a group that contains the information you want
6. The order of the results so a specific row among duplicates can be returned.

A SELECT statement looks like this:

```
SELECT column names
FROM table or view name
WHERE search condition
GROUP BY column names
HAVING search condition
ORDER BY column-name
```

The SELECT and FROM clauses must be specified. The other clauses are optional.

With the SELECT clause, you specify the name of each column you want to retrieve. For example:

```
SELECT EMPNO, LASTNAME, WORKDEPT
:
:
```

You can specify that only one column be retrieved, or as many as 8000 columns. The value of each column you name is retrieved in the order specified in the SELECT clause.

If you want to retrieve all columns (in the same order as they appear in the table's definition), use an asterisk (*) instead of naming the columns:

```
SELECT *
:
:
```

The FROM clause specifies the table that you want to select data *from*. You can select columns from more than one table. When issuing a SELECT, you must specify a FROM clause. Issue the following statement:

```
SELECT *
FROM EMPLOYEE
```

The result is all of the columns and rows from table EMPLOYEE.

The SELECT list can also contain expressions, including constants, special registers, and scalar subselects. An AS clause can also be used to give the resulting column a name. For example, issue the following statement:

```
SELECT LASTNAME, SALARY * .05 AS RAISE
FROM EMPLOYEE
WHERE EMPNO = '200140'
```

The result of this statement is:

Table 4. Results for query

LASTNAME	RAISE
NATZ	1421

If SQL is unable to find a row that satisfies the search condition, an SQLCODE of +100 is returned.

If SQL finds errors while running your select-statement, a negative SQLCODE is returned. If SQL finds more host variables than results, +326 is returned.

For information about qualifying the data returned, see "Specifying a search condition using the WHERE clause" on page 63.

Specifying a search condition using the WHERE clause

The WHERE clause specifies a search condition that identifies the row or rows you want to retrieve, update, or delete. The number of rows you process with an SQL statement then depends on the number of rows that satisfy the WHERE clause **search condition**. A search condition consists of one or more **predicates**. A predicate specifies a test that you want SQL to apply to a specified row or rows of a table. For more information about predicates, see “Performing complex search conditions” on page 75.

In the following example, WORKDEPT = 'C01' is a predicate, WORKDEPT and 'C01' are expressions, and the equal sign (=) is a comparison operator. Note that character values are enclosed in apostrophes (''); numeric values are not. This applies to all constant values wherever they are coded within an SQL statement. For example, to specify that you are interested in the rows where the department number is C01, you would say:

```
... WHERE WORKDEPT = 'C01'
```

In this case, the search condition consists of one predicate: WORKDEPT = 'C01'.

To further illustrate WHERE, put it into a SELECT statement. Assume that each department listed in the CORPDATA.DEPARTMENT table has a unique department number. You want to retrieve the department name and manager number from the CORPDATA.DEPARTMENT table for department C01. Issue the following statement:

```
SELECT DEPTNAME, MGRNO
FROM CORPDATA.DEPARTMENT
WHERE DEPTNO = 'C01'
```

When this statement is run, the result is one row:

Table 5. Result table

DEPTNAME	MGRNO
INFORMATION CENTER	000030

If the search condition contains character or UCS-2 graphic column predicates, the sort sequence that is in effect when the query is run is applied to those predicates. See Chapter 8, “Sort sequences in SQL” on page 115 for more information about sort sequence and selection. If a sort sequence is not being used, character constants must be specified in upper or lower case to match the column or expression they are being compared to.

For more details about using the WHERE clause, see the following sections:

- “Expressions in the WHERE Clause”
- “Comparison operators” on page 65
- “NOT Keyword” on page 65

Expressions in the WHERE Clause

An expression in a WHERE clause names or specifies something you want to compare to something else. Each expression, when evaluated by SQL, is a character string, date/time/timestamp, or a numeric value. The expressions you specify can be:

- A **column name** names a column. For example:

```
... WHERE EMPNO = '000200'
```

EMPNO names a column that is defined as a 6-byte character value. Equality comparisons (that is, $X = Y$ or $X <> Y$) can be performed on character data. Other types of comparisons can also be evaluated for character data.

However, you cannot compare character strings to numbers. You also cannot perform arithmetic operations on character data (even though *EMPNO* is a character string that appears to be a number). A cast function can be used to convert character and numeric data into values that can be compared. You can add and subtract date/time values and durations.

- An **expression** identifies two values that are added (+), subtracted (-), multiplied (*), divided (/), have exponentiation (**), or concatenated (CONCAT or ||) to result in a value. The operands of an expression can be:
 - A constant
 - A column
 - A host variable
 - A value returned from a function
 - A special register
 - A subquery
 - Another expression

For example:

```
... WHERE INTEGER(PRENDATE - PRSTDATE) > 100
```

When the order of evaluation is not specified by parentheses, the expression is evaluated in the following order:

1. Prefix operators
2. Exponentiation
3. Multiplication, division, and concatenation
4. Addition and subtraction

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

- A **constant** specifies a literal value for the expression. For example:

```
... WHERE 40000 < SALARY
```

SALARY names a column that is defined as an 9-digit packed decimal value (DECIMAL(9,2)). It is compared to the numeric constant 40000.
- A **host variable** identifies a variable in an application program. For example:

```
... WHERE EMPNO = :EMP
```
- A **special register** identifies a special value defined by the database manager. For example:

```
... WHERE LASTNAME = USER
```
- The **NULL** value specifies the condition of having an unknown value.

```
... WHERE DUE_DATE IS NULL
```
- A subquery. For details about using subqueries, see Chapter 7, “Using Subqueries” on page 105.

A search condition need not be limited to two column names or constants separated by arithmetic or comparison operators. You can develop a complex search condition that specifies several predicates separated by AND and OR. No

matter how complex the search condition, it supplies a TRUE or FALSE value when evaluated against a row. There is also an *unknown* truth value, which is effectively false. That is, if the value of a row is null, this null value is not returned as a result of a search because it is not less than, equal to, or greater than the value specified in the search condition. More complex search conditions and predicates are described in “Performing complex search conditions” on page 75.

To fully understand the WHERE clause, you need to know how SQL evaluates search conditions and predicates, and compares the values of expressions. This topic is discussed in the SQL Reference book.

Comparison operators

SQL supports the following comparison operators:

=	Equal to
<> or \neq or !=	Not equal to
<	Less than
>	Greater than
<= or \geq or !>	Less than or equal to (or not greater than)
>= or \leq or !<	Greater than or equal to (or not less than)

NOT Keyword

You can precede a predicate with the NOT keyword to specify that you want the opposite of the predicate’s value (that is, TRUE if the predicate is FALSE, or vice versa). NOT applies only to the predicate it precedes, not to all predicates in the WHERE clause. For example, to indicate that you are interested in all employees except those working in department C01, you could say:

```
... WHERE NOT WORKDEPT = 'C01'
```

which is equivalent to:

```
... WHERE WORKDEPT <> 'C01'
```

GROUP BY clause

Without a GROUP BY clause, the application of SQL column functions returns *one* row. When GROUP BY is used, the function is applied to *each* group, thereby returning as many rows as there are groups.

The GROUP BY clause allows you to find the characteristics of groups of rows rather than individual rows. When you specify a GROUP BY clause, SQL divides the selected rows into groups such that the rows of each group have matching values in one or more columns or expressions. Next, SQL processes each group to produce a single-row result for the group. You can specify one or more columns or expressions in the GROUP BY clause to group the rows. The items you specify in the SELECT statement are properties of each group of rows, not properties of individual rows in a table or view.

For example, the CORPDATA.EMPLOYEE table has several sets of rows, and each set consists of rows describing members of a specific department. To find the average salary of people in each department, you could issue:

```
SELECT WORKDEPT, DECIMAL (AVG(SALARY),5,0)
FROM CORPDATA.EMPLOYEE
GROUP BY WORKDEPT
```

The result is several rows, one for each department.

WORKDEPT	AVG-SALARY
A00	40850
B01	41250
C01	29722
D11	25147
D21	25668
E01	40175
E11	21020
E21	24086

Notes:

1. Grouping the rows does not mean ordering them. Grouping puts each selected row in a group, which SQL then processes to derive characteristics of the group. Ordering the rows puts all the rows in the results table in ascending or descending collating sequence. (“ORDER BY clause” on page 68 describes how to do this.) Depending on the implementation selected by the database manager, the resulting groups may appear to be ordered.
2. If there are null values in the column you specify in the GROUP BY clause, a single-row result is produced for the data in the rows with null values.
3. If the grouping occurs over character or UCS-2 graphic columns, the sort sequence in effect when the query is run is applied to the grouping. See Chapter 8, “Sort sequences in SQL” on page 115 for more information about sort sequence and selection.

When you use GROUP BY, you list the columns or expressions you want SQL to use to group the rows. For example, suppose you want a list of the number of people working on each major project described in the CORPDATA.PROJECT table. You could issue:

```
SELECT SUM(PRSTAFF), MAJPROJ
FROM CORPDATA.PROJECT
GROUP BY MAJPROJ
```

The result is a list of the company’s current major projects and the number of people working on each project:

SUM(PRSTAFF)	MAJPROJ
6	AD3100
5	AD3110
10	MA2100
8	MA2110
5	OP1000
4	OP2000
3	OP2010
32.5	?

You can also specify that you want the rows grouped by more than one column or expression. For example, you could issue a select-statement to find the average salary for men and women in each department, using the CORPDATA.EMPLOYEE table. To do this, you could issue:

```
SELECT WORKDEPT, SEX, DECIMAL(AVG(SALARY),5,0) AS AVG_WAGES
FROM CORPDATA.EMPLOYEE
GROUP BY WORKDEPT, SEX
```

Results in:

WORKDEPT	SEX	AVG_WAGES
A00	F	49625
A00	M	35000
B01	M	41250
C01	F	29722
D11	F	25817
D11	M	24764
D21	F	26933
D21	M	24720
E01	M	40175
E11	F	22810
E11	M	16545
E21	F	25370
E21	M	23830

Because you did not include a WHERE clause in this example, SQL examines and process all rows in the CORPDATA.EMPLOYEE table. The rows are grouped first by department number and next (within each department) by sex before SQL derives the average SALARY value for each group.

HAVING clause

You can use the HAVING clause to specify a search condition for the groups selected based on a GROUP BY clause. The HAVING clause says that you want *only* those groups that satisfy the condition in that clause. Therefore, the search condition you specify in the HAVING clause must test properties of each group rather than properties of individual rows in the group.

The HAVING clause follows the GROUP BY clause and can contain the same kind of search condition you can specify in a WHERE clause. In addition, you can specify column functions in a HAVING clause. For example, suppose you wanted to retrieve the average salary of women in each department. To do this, you would use the AVG column function and group the resulting rows by WORKDEPT and specify a WHERE clause of SEX = 'F'.

To specify that you want this data only when all the female employees in the selected department have an education level equal to or greater than 16 (a college graduate), use the HAVING clause. The HAVING clause tests a property of the group. In this case, the test is on MIN(EDLEVEL), which is a group property:

```

SELECT WORKDEPT, DECIMAL(AVG(SALARY),5,0) AS AVG_WAGES, MIN(EDLEVEL) AS MIN_EDUC
  FROM CORPDATA.EMPLOYEE
 WHERE SEX='F'
 GROUP BY WORKDEPT
 HAVING MIN(EDLEVEL)>=16

```

Results in:

WORKDEPT	AVG_WAGES	MIN_EDUC
A00	49625	18
C01	29722	16
D11	25817	17

You can use multiple predicates in a HAVING clause by connecting them with AND and OR, and you can use NOT for any predicate of a search condition.

Note: If you intend to update a column or delete a row, you cannot include a GROUP BY or HAVING clause in the SELECT statement within a DECLARE CURSOR statement. (The DECLARE CURSOR statement is described in Chapter 9, “Using a Cursor” on page 121.) These clauses make it a read-only cursor.

Predicates with arguments that are not column functions can be coded in either WHERE or HAVING clauses. It is usually more efficient to code the selection criteria in the WHERE clause because it is handled earlier in the query processing. The HAVING selection is performed in post processing of the result table.

If the search condition contains predicates involving character or UCS-2 graphic columns, the sort sequence in effect when the query is run is applied to those predicates. See Chapter 8, “Sort sequences in SQL” on page 115 for more information about sort sequence and selection.

ORDER BY clause

You can specify that you want selected rows returned in a particular order, sorted by ascending or descending collating sequence of a column’s or expression’s value, with the ORDER BY clause. You can use an ORDER BY clause as you would a GROUP BY clause: specify the columns or expressions you want SQL to use when retrieving the rows in a collated sequence.

For example, to retrieve the names and department numbers of female employees listed in the alphanumeric order of their department numbers, you could use this select-statement:

```

SELECT LASTNAME, WORKDEPT
  FROM CORPDATA.EMPLOYEE
 WHERE SEX='F'
 ORDER BY WORKDEPT

```

Results in:

LASTNAME	WORKDEPT
HAAS	A00
HEMMINGER	A00
KWAN	C01

LASTNAME	WORKDEPT
QUINTANA	C01
NICHOLLS	C01
NATZ	C01
PIANKA	D11
SCOUTTEN	D11
LUTZ	D11
JOHN	D11
PULASKI	D21
JOHNSON	D21
PEREZ	D21
HENDERSON	E11
SCHNEIDER	E11
SETRIGHT	D11
SCHWARTZ	E11
SPRINGER	E11
WONG	E21

Note: Null values are ordered as the highest value.

The column specified in the ORDER BY clause does not need to be included in the SELECT clause. For example, the following statement will return all female employees ordered with the largest salary first:

```
SELECT LASTNAME, FIRSTNAME
       FROM CORPDATA.EMPLOYEE
       WHERE SEX='F'
       ORDER BY SALARY DESC
```

If an AS clause is specified to name a result column in the select-list, this name can be specified in the ORDER BY clause. The name specified in the AS clause must be unique in the select-list. For example, to retrieve the full name of employees listed in alphabetic order, you could use this select-statement:

```
SELECT LASTNAME CONCAT FIRSTNAME AS FULLNAME
       FROM CORPDATA.EMPLOYEE
       ORDER BY FULLNAME
```

This select-statement could optionally be written as:

```
SELECT LASTNAME CONCAT FIRSTNAME
       FROM CORPDATA.EMPLOYEE
       ORDER BY LASTNAME CONCAT FIRSTNAME
```

Instead of naming the columns to order the results, you can use a number. For example, ORDER BY 3 specifies that you want the results ordered by the *third* column of the results table, as specified by the select-list. Use a number to order the rows of the results table when the sequencing value is not a named column.

You can also specify whether you want SQL to collate the rows in ascending (ASC) or descending (DESC) sequence. An ascending collating sequence is the default. In the previous select-statement, SQL first returns the row with the lowest

FULLNAME expression (alphabetically and numerically), followed by rows with higher values. To order the rows in descending collating sequence based on this name, specify:

```
... ORDER BY FULLNAME DESC
```

As with *GROUP BY*, you can specify a secondary ordering sequence (or several levels of ordering sequences) as well as a primary one. In the previous example, you might want the rows ordered first by department number, and within each department, ordered by employee name. To do this, specify:

```
... ORDER BY WORKDEPT, FULLNAME
```

If character columns or UCS-2 graphic columns are used in the *ORDER BY* clause, ordering for these columns is based on the sort sequence in effect when the query is run. See Chapter 8, “Sort sequences in SQL” on page 115 for more information about sort sequence and its affect on ordering.

Static **SELECT** statements

For a static *SELECT* statement (one embedded in an SQL program), an *INTO* clause must be specified before the *FROM* clause. The *INTO* clause names the host variables (variables in your program used to contain retrieved column values). The value of the first result column specified in the *SELECT* clause is put into the first host variable named in the *INTO* clause; the second value is put into the second host variable, and so on.

The result table for a *SELECT INTO* should contain just one row. For example, each row in the *CORPDATA.EMPLOYEE* table has a unique *EMPNO* (employee number) column. The result of a *SELECT INTO* statement for this table if the *WHERE* clause contains an equal comparison on the *EMPNO* column, would be exactly one row (or no rows). Finding more than one row is an error, but one row is still returned. You can control which row will be returned in this error condition by specifying the *ORDER BY* clause. If you use the *ORDER BY* clause, the first row in the result table is returned.

If you want more than one row to be the result of a *SELECT INTO* statement, use a *DECLARE CURSOR* statement to select the rows, followed by a *FETCH* statement to move the column values into host variables one or many rows at a time. Using cursors is described in Chapter 9, “Using a Cursor” on page 121.

When using the select-statement in an application program, list the column names to give your program more data independence. There are two reasons for this:

1. When you look at the source code statement, you can easily see the one-to-one correspondence between the column names in the *SELECT* clause and the host variables named in the *INTO* clause.
2. If a column is added to a table or view you access and you use “*SELECT * ...*,” and you create the program again from source, the *INTO* clause does not have a matching host variable named for the new column. The extra column causes you to get a warning (not an error) in the *SQLCA* (*SQLWARN4* will contain a “W”).

Null Values to indicate absence of column values in a row

A *NULL* value indicates the absence of a column value in a row. A null value is not the same as zero or all blanks. A null value is the same as “unknown”. Null values can be used as a condition in the *WHERE* and *HAVING* clauses, and as a mathematical argument. For example, a *WHERE* clause can specify a column that, for some rows, contains a null value. Normally, a comparison predicate using a column that contains null values does not select a row that has a null value for the column. This is because a null value is neither less than, equal to, nor greater than the value specified in the condition. To select the values for all rows that contain a null value for the manager number, you could specify:

```
SELECT DEPTNO, DEPTNAME, ADMRDEPT
FROM CORPDATA.DEPARTMENT
WHERE MGRNO IS NULL
```

The result would be:

DEPTNO	DEPTNAME	ADMRDEPT
D01	DEVELOPMENT CENTER	A00
F22	BRANCH OFFICE F2	E01
G22	BRANCH OFFICE G2	E01
H22	BRANCH OFFICE H2	E01
I22	BRANCH OFFICE I2	E01
J22	BRANCH OFFICE J2	E01

To get the rows that do not have a null value for the manager number, you could change the *WHERE* clause like this:

```
WHERE MGRNO IS NOT NULL
```

For more information about the use of null values, see the SQL Reference book.

Special registers in SQL statements

You can specify certain “special registers” in SQL statements. For *locally* run SQL statements, the special registers and their contents are shown in the following table:

Special Registers	Contents
CURRENT DATE CURRENT_DATE	The current date.
CURRENT PATH CURRENT_PATH CURRENT FUNCTION PATH	The SQL path used to resolve unqualified data type names, procedure names, and function names in dynamically prepared SQL statements.
CURRENT SCHEMA	The schema name used to qualify unqualified database object references where applicable in dynamically prepared SQL statements.
CURRENT SERVER CURRENT_SERVER	The name of the relational database currently being used.
CURRENT TIME CURRENT_TIME	The current time.

Special Registers	Contents
CURRENT_TIMESTAMP CURRENT_TIMESTAMP	The current date and time in timestamp format.
CURRENT_TIMEZONE CURRENT_TIMEZONE	A duration of time that links local time to Universal Time Coordinated (UTC) using the formula: local time - CURRENT_TIMEZONE = UTC It is taken from the system value QUTCOFFSET.
USER	The run-time authorization identifier (user profile) of the job.

If a single statement contains more than one reference to any of CURRENT DATE, CURRENT TIME, or CURRENT_TIMESTAMP special registers, or the CURDATE, CURTIME, or NOW scalar functions, all values are based on a single clock reading.

For *remotely* run SQL statements, the special registers and their contents are shown in the following table:

Special Registers	Contents
CURRENT_DATE CURRENT_DATE CURRENT_TIME CURRENT_TIME CURRENT_TIMESTAMP CURRENT_TIMESTAMP	The current date and time at the remote system, not the local system.
CURRENT_TIMEZONE CURRENT_TIMEZONE	A duration of time that links the remote system time to UTC.
CURRENT_SERVER CURRENT_SERVER	The name of the relational database currently being used.
CURRENT_SCHEMA	The current schema value at the remote system.
USER	The run-time authorization identifier of the server job on the remote system.
CURRENT_PATH CURRENT_PATH CURRENT_FUNCTION_PATH	The current path value at the remote system.

When a query over a distributed table references a special register, the contents of the special register on the system that requests the query are used. For more information about distributed tables, see DB2 Multisystem book.

Casting data types

Sometimes you will find situations where the *type* of a data type needs to be cast, or changed, to a different data type or to the same data type with a different length, precision, or scale. For example, if you wanted to compare two columns of different types, such as a char and an integer, you can change the char to an

integer or the integer to a char to make the comparison possible. A data type that can be changed to another data type is *castable* from the source data type to the target data type.

You can use cast functions or CAST specifications to explicitly cast a data type to another data type. For example, if you have a column of dates (BIRTHDATE) defined as DATE and wanted to cast the column data type to CHARACTER with a fixed length of 10, you would enter the following:

```
SELECT CHAR (BIRTHDATE,USA)
FROM CORPDATA.EMPLOYEE
```

You can also use the CAST function to cast data types directly.

```
SELECT CAST(BIRTHDATE AS CHAR(10))
FROM CORPDATA.EMPLOYEE
```

For more details about casting data types, see Casting between data types in the SQL Reference topic.

Date, Time, and Timestamp data types

Date, time, and timestamp are data types represented in an internal form not seen by the SQL user. Date, time, and timestamp can be represented by character string values and assigned to character string variables. The database manager recognizes the following as date, time, and timestamp values:

- A value returned by the DATE, TIME, or TIMESTAMP scalar functions.
- A value returned by the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers.
- A character string when it is an operand of an arithmetic expression or a comparison *and* the other operand is a date, time, or timestamp. For example, in the predicate:

```
... WHERE HIREDATE < '1950-01-01'
```

if HIREDATE is a date column, the character string '1950-01-01' is interpreted as a date.

- A character string variable or constant used to set a date, time, or timestamp column in either the SET clause of an UPDATE statement, or the VALUES clause of an INSERT statement.

For more information about character string formats of date, time, and timestamp values, see Datetime Values in the SQL Reference book .

See also the following topics:

- "Specifying current date and time values"
- "Date/Time arithmetic" on page 74

Specifying current date and time values

You can specify a current date, time, or timestamp in an expression by specifying one of three special registers: CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The value of each is based on a time-of-day clock reading obtained during the running of the statement. Multiple references to CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP within the same SQL statement use the same value. The following statement returns the age (in years) of each employee in the EMPLOYEE table when the statement is run:

```
SELECT YEAR(CURRENT DATE - BIRTHDATE)
FROM CORPDATA.EMPLOYEE
```

The CURRENT TIMEZONE special register allows a local time to be converted to Universal Time Coordinated (UTC). For example, if you have a table named DATETIME, containing a time column type with a name of STARTT, and you want to convert STARTT to UTC, you can use the following statement:

```
SELECT STARTT - CURRENT TIMEZONE
FROM DATETIME
```

Date/Time arithmetic

Addition and subtraction are the only arithmetic operators applicable to date, time, and timestamp values. You can increment and decrement a date, time, or timestamp by a duration; or subtract a date from a date, a time from a time, or a timestamp from a timestamp. For a detailed description of date and time arithmetic, see Datetime arithmetic in the SQL Reference book.

Preventing duplicate rows

When SQL evaluates a select-statement, several rows might qualify to be in the result table, depending on the number of rows that satisfy the select-statement's search condition. Some of the rows in the result table might be duplicates. You can specify that you do not want any duplicates by using the DISTINCT keyword, followed by the list of column names:

```
SELECT DISTINCT JOB, SEX
...
```

DISTINCT means you want to select only the unique rows. If a selected row duplicates another row in the result table, the duplicate row is ignored (it is not put into the result table). For example, suppose you want a list of employee job codes. You do not need to know which employee has what job code. Because it is probable that several people in a department have the same job code, you can use DISTINCT to ensure that the result table has only unique values.

The following example shows how to do this:

```
SELECT DISTINCT JOB
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D11'
```

The result is two rows:

JOB
DESIGNER
MANAGER

If you do not include DISTINCT in a SELECT clause, you might find duplicate rows in your result, because SQL returns the JOB column's value for each row that satisfies the search condition. Null values are treated as duplicate rows for DISTINCT.

If you include DISTINCT in a SELECT clause and you also include a shared-weight sort sequence, fewer values are returned. The sort sequence causes values that contain the same characters to be weighted the same. If 'MGR', 'Mgr',

and 'mgr' were all in the same table, only one of these values would be returned. See Chapter 8, "Sort sequences in SQL" on page 115 for more information about sort sequence and selection.

Performing complex search conditions

In addition to the basic comparison predicates (=, >, <, etc), a search condition can contain any of the keywords BETWEEN, IN, EXISTS, IS NULL, and LIKE. A search condition can also include a subquery. See Chapter 7, "Using Subqueries" on page 105 for more information and examples.

For character and UCS-2 graphic column predicates, the sort sequence is applied to the operands before evaluation of the predicates for BETWEEN, IN, EXISTS, and LIKE clauses. See Chapter 8, "Sort sequences in SQL" on page 115 for more information about the using sort sequence with selection.

You can also perform multiple search conditions. See "Multiple search conditions within a WHERE clause" on page 77 for more information.

- **BETWEEN ... AND ...** is used to specify a search condition that is satisfied by any value that falls on or between two other values. For example, to find all employees who were hired in 1987, you could use this:

```
... WHERE HIREDATE BETWEEN '1987-01-01' AND '1987-12-31'
```

The BETWEEN keyword is inclusive. A more complex, but explicit, search condition that produces the same result is:

```
... WHERE HIREDATE >= '1987-01-01' AND HIREDATE <= '1987-12-31'
```

- **IN** says you are interested in rows in which the value of the specified expression is among the values you listed. For example, to find the names of all employees in departments A00, C01, and E21, you could specify:

```
... WHERE WORKDEPT IN ('A00', 'C01', 'E21')
```

- **EXISTS** says you are interested in testing for the existence of certain rows. For example, to find out if there are any employees that have a salary greater than 60000, you could specify:

```
EXISTS (SELECT * FROM EMPLOYEE WHERE SALARY > 60000)
```

- **IS NULL** says that you are interested in testing for null values. For example, to find out if there are any employees without a phone listing, you could specify:

```
... WHERE EMPLOYEE.PHONE IS NULL
```

- **LIKE** says you are interested in rows in which a column value is similar to the value you supply. When you use LIKE, SQL searches for a character string similar to the one you specify. The degree of similarity is determined by two special characters used in the string that you include in the search condition:

— An underline character stands for any single character.

% A percent sign stands for an unknown string of 0 or more characters. If the percent sign starts the search string, then SQL allows 0 or more character(s) to precede the matching value in the column. Otherwise, the search string must begin in the first position of the column.

Note: If you are operating on MIXED data, the following distinction applies: an SBCS underline character refers to one SBCS character. No such restriction applies to the percent sign; that is, a percent sign refers to any number of SBCS or DBCS characters. See the SQL Reference book for more information about the LIKE predicate and MIXED data.

Use the underline character or percent sign either when you do not know or do not care about all the characters of the column's value. For example, to find out which employees live in Minneapolis, you could specify:

```
... WHERE ADDRESS LIKE '%MINNEAPOLIS%'
```

SQL returns any row with the string MINNEAPOLIS in the ADDRESS column, no matter where the string occurs.

In another example, to list the towns whose names begin with 'SAN', you could specify:

```
... WHERE TOWN LIKE 'SAN%'
```

If you want to find any addresses where the street name isn't in your master street name list, you can use an expression in the LIKE expression. In this example, the STREET column in the table is assumed to be upper case.

```
... WHERE UCASE (:address_variable) NOT LIKE '%||STREET||%'
```

If you want to search for a character string that contains either the underscore or percent character, use the ESCAPE clause to specify an escape character. For example, to see all businesses that have a percent in their name, you could specify:

```
... WHERE BUSINESS_NAME LIKE '%@%' ESCAPE '@'
```

The first and last percent characters are interpreted as usual. The combination '@%' is taken as the actual percent character. See "Special considerations for LIKE" for more details.

For a complete listing of predicates, see Predicates in the SQL Reference topic.

Special considerations for LIKE

- When host variables are used in place of string constants in a search pattern, you should consider using varying length host variables. This allows you to:
 - Assign previously used string constants to host variables without any change.
 - Obtain the same selection criteria and results as if a string constant was used.
- When fixed-length host variables are used in place of string constants in a search pattern, you should ensure that the value specified in the host variable matches the pattern previously used by the string constants. All characters in a host variable that are not assigned a value are initialized with a blank.

For example, if you did a search using the string pattern 'ABC%' in a *varying length* host variable, these are some of the values that could be returned:

```
'ABCD      ' 'ABCDE'   'ABCxxx'   'ABC      '
```

However, if you did a search using the search pattern 'ABC%' contained in a host variable with a *fixed length* of 10, these are some the values that could be returned assuming the column has a length of 12:

```
'ABCDE      ' 'ABCD      ' 'ABCxxx    ' 'ABC      ' '          '
```

Note that all returned values start with 'ABC' and end with at least six blanks. This is because the last six characters in the host variable were not assigned a specific value so blanks were used.

If you wanted to do a search using a fixed-length host variable where the last 7 characters could be anything, you would search for 'ABC%%%%%%%%'. These are some values that could be returned:

'ABCDEFGHJIJ' 'ABCXXXXXX' 'ABCDE' 'ABCDD'

Multiple search conditions within a WHERE clause

In the section “Specifying a search condition using the WHERE clause” on page 63, you saw how to use one search condition. You can qualify your request further by coding a search condition that includes several predicates. The search condition you specify can contain any of the comparison operators or the predicates BETWEEN, IN, LIKE, EXISTS, IS NULL, and IS NOT NULL.

You can combine any two predicates with the connectors AND and OR. In addition, you can use the NOT keyword to specify that the desired search condition is the negated value of the specified search condition. A WHERE clause can have as many predicates as you want.

- **AND** says that, for a row to qualify, the row must satisfy both predicates of the search condition. For example, to find out which employees in department D21 were hired after December 31, 1987, you would specify:

```
...  
WHERE WORKDEPT = 'D21' AND HIREDATE > '1987-12-31'
```

- **OR** says that, for a row to qualify, the row can satisfy the condition set by either or both predicates of the search condition. For example, to find out which employees are in either department C01 or D11, you could specify :

```
...  
WHERE WORKDEPT = 'C01' OR WORKDEPT = 'D11'
```

Note: You could also use IN to specify this request: WHERE WORKDEPT IN ('C01', 'D11').

- **NOT** says that, to qualify, a row must not meet the criteria set by the search condition or predicate that follows the NOT. For example, to find all employees in department E11 except those with a job code equal to analyst, you could specify:

```
...  
WHERE WORKDEPT = 'E11' AND NOT JOB = 'ANALYST'
```

When SQL evaluates search conditions that contain these connectors, it does so in a specific order. SQL first evaluates the NOT clauses, next evaluates the AND clauses, and then the OR clauses.

You can change the order of evaluation by using parentheses. The search conditions enclosed in parentheses are evaluated first. For example, to select all employees in departments E11 and E21 who have education levels greater than 12, you could specify:

```
...  
WHERE EDLEVEL > 12 AND  
      (WORKDEPT = 'E11' OR WORKDEPT = 'E21')
```

The parentheses determine the meaning of the search condition. In this example, you want all rows that have a:

WORKDEPT value of E11 or E21, and
EDLEVEL value greater than 12

If you did not use parentheses:

```
...  
WHERE EDLEVEL > 12 AND WORKDEPT = 'E11'  
      OR WORKDEPT = 'E21'
```

Your result is different. The selected rows are rows that have:

WORKDEPT = E11 and EDLEVEL > 12, or

WORKDEPT = E21, regardless of the EDLEVEL value

Joining data from more than one table

Sometimes the information you want to see is not in a single table. To form a row of the result table, you might want to retrieve some column values from one table and some column values from another table. You can retrieve and join column values from two or more tables into a single row.

Several different types of joins are supported by DB2 UDB for iSeries: inner join, left outer join, right outer join, left exception join, right exception join, and cross join.

- An “Inner Join” on page 79 returns only the rows from each table that have matching values in the join columns. Any rows that do not have a match between the tables will not appear in the result table.
- A “Left Outer Join” on page 80 returns values for all of the rows from the first table (the table on the left) and the values from the second table for the rows that match. Any rows that do not have a match in the second table will return the null value for all columns from the second table.
- A “Right Outer Join” on page 80 return values for all of the rows from the second table (the table on the right) and the values from the first table for the rows that match. Any rows that do not have a match in the first table will return the null value for all columns from the first table.
- A Left Exception Join returns only the rows from the left table that do not have a match in the right table. Columns in the result table that come from the right table have the null value.
- A Right Exception Join returns only the rows from the right table that do not have a match in the left table. Columns in the result table that come from the left table have the null value.
- A “Cross Join” on page 81 returns a row in the result table for each combination of rows from the tables being joined (a Cartesian Product).

| You can simulate a Full Outer Join using a Left Outer join and a Right Exception
| Join. See “Simulating a Full Outer Join” on page 82 for details. Additionally, you
| can use multiple join types in a single statement. See “Multiple join types in one
| statement” on page 82 for details.

Notes on joins

When you join two or more tables:

- If there are common column names, you must qualify each common name with the name of the table (or a correlation name). Column names that are unique do not need to be qualified.
- If you do not list the column names you want, but instead use `SELECT *`, SQL returns rows that consist of all the columns of the first table, followed by all the columns of the second table, and so on.
- You must be authorized to select rows from each table or view specified in the `FROM` clause.
- The sort sequence is applied to all character and UCS-2 graphic columns being joined.

Inner Join

With an inner join, column values from one row of a table are combined with column values from another row of another (or the same) table to form a single row of data. SQL examines both tables specified for the join to retrieve data from all the rows that meet the search condition for the join. There are two ways of specifying an inner join: using the JOIN syntax, and using the WHERE clause.

Suppose you want to retrieve the employee numbers, names, and project numbers for all employees that are responsible for a project. In other words, you want the *EMPNO* and *LASTNAME* columns from the *CORPDATA.EMPLOYEE* table and the *PROJNO* column from the *CORPDATA.PROJECT* table. Only employees with last names starting with 'S' or later should be considered. To find this information, you need to join the two tables.

Inner join using JOIN syntax

To use the inner join syntax, both of the tables you are joining are listed in the FROM clause, along with the join condition that applies to the tables. The join condition is specified after the ON keyword and determines how the two tables are to be compared to each other to produce the join result. The condition can be any comparison operator; it does not need to be the equal operator. Multiple join conditions can be specified in the ON clause separated by the AND keyword. Any additional conditions that do not relate to the actual join are specified in either the WHERE clause or as part of the actual join in the ON clause.

```
SELECT EMPNO, LASTNAME, PROJNO
FROM CORPDATA.EMPLOYEE INNER JOIN CORPDATA.PROJECT
ON EMPNO = RESPEMP
WHERE LASTNAME > 'S'
```

In this example, the join is done on the two tables using the *EMPNO* and *RESPEMP* columns from the tables. Since only employees that have last names starting with at least 'S' are to be returned, this additional condition is provided in the WHERE clause.

This query returns the following output:

EMPNO	LASTNAME	PROJNO
000250	SMITH	AD3112
000060	STERN	MA2110
000100	SPENSER	OP2010
000020	THOMPSON	PL2100

Inner join using the WHERE clause

Using the WHERE clause to perform this same join is written with both the join condition and the additional selection condition in the WHERE clause. The tables to be joined are listed in the FROM clause, separated by commas.

```
SELECT EMPNO, LASTNAME, PROJNO
FROM CORPDATA.EMPLOYEE, CORPDATA.PROJECT
WHERE EMPNO = RESPEMP
AND LASTNAME > 'S'
```

This query returns the same output as the previous example.

Left Outer Join

A left outer join will return all the rows that an inner join returns plus one row for each of the other rows in the first table that did not have a match in the second table.

Suppose you want to find all employees and the projects they are currently responsible for. You want to see those employees that are not currently in charge of a project as well. The following query will return a list of all employees whose names are greater than 'S', along with their assigned project numbers.

```
SELECT EMPNO, LASTNAME, PROJNO
FROM CORPDATA.EMPLOYEE LEFT OUTER JOIN CORPDATA.PROJECT
ON EMPNO = RESPEMP
WHERE LASTNAME > 'S'
```

The result of this query contains some employees that do not have a project number. They are listed in the query, but have the null value returned for their project number.

EMPNO	LASTNAME	PROJNO
000020	THOMPSON	PL2100
000060	STERN	MA2110
000100	SPENSER	OP2010
000170	YOSHIMURA	-
000180	SCOUTTEN	-
000190	WALKER	-
000250	SMITH	AD3112
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-
200170	YAMAMOTO	-
200280	SCHWARTZ	-
200310	SPRINGER	-
200330	WONG	-

Notes

Using the RRN scalar function to return the relative record number for a column in the table on the right in a left outer join or exception join will return a value of 0 for the unmatched rows.

Right Outer Join

A right outer join will return all the rows that an inner join returns plus one row for each of the other rows in the second table that did not have a match in the first table. It is the same as a left outer join with the tables specified in the opposite order.

The query that was used as the left outer join example could be rewritten as a right outer join as follows:

```
SELECT EMPNO, LASTNAME, PROJNO
FROM CORPDATA.PROJECT RIGHT OUTER JOIN CORPDATA.EMPLOYEE
ON EMPNO = RESPEMP
WHERE LASTNAME > 'S'
```

The results of this query are identical to the results from the left outer join query.

Exception Join

A left exception join returns only the rows from the first table that do NOT have a match in the second table. Using the same tables as before, return those employees that are not responsible for any projects.

```
SELECT EMPNO, LASTNAME, PROJNO
FROM CORPDATA.EMPLOYEE EXCEPTION JOIN CORPDATA.PROJECT
ON EMPNO = RESPEMP
WHERE LASTNAME > 'S'
```

This join returns the output:

EMPNO	LASTNAME	PROJNO
000170	YOSHIMURA	-
000180	SCOUTTEN	-
000190	WALKER	-
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-
200170	YAMAMOTO	-
200280	SCHWARTZ	-
200310	SPRINGER	-
200330	WONG	-

An exception join can also be written as a subquery using the NOT EXISTS predicate. The previous query could be rewritten in the following way:

```
SELECT EMPNO, LASTNAME
FROM CORPDATA.EMPLOYEE
WHERE LASTNAME > 'S'
AND NOT EXISTS
(SELECT * FROM CORPDATA.PROJECT
WHERE EMPNO = RESPEMP)
```

The only difference in this query is that it cannot return values from the PROJECT table.

There is a right exception join, too, that works just like a left exception join but with the tables reversed.

Cross Join

A cross join (or Cartesian Product join) will return a result table where each row from the first table is combined with each row from the second table. The number of rows in the result table is the product of the number of rows in each table. If the tables involved are large, this join can take a very long time.

A cross join can be specified in two ways: using the JOIN syntax or by listing the tables in the FROM clause separated by commas without using a WHERE clause to supply join criteria.

Suppose the following tables exist.

Table 6. Table A

ACOL1	ACOL2
A1	AA1
A2	AA2
A3	AA3

Table 7. Table B

BCOL1	BCOL2
B1	BB1
B2	BB2

The following two select statements produce identical results.

```
SELECT * FROM A CROSS JOIN B
SELECT * FROM A, B
```

The result table for either of these select statements looks like this:

ACOL1	ACOL2	BCOL1	BCOL2
A1	AA1	B1	BB1
A1	AA1	B2	BB2
A2	AA2	B1	BB1
A2	AA2	B2	BB2
A3	AA3	B1	BB1
A3	AA3	B2	BB2

Simulating a Full Outer Join

Like the left and right outer joins, a full outer join returns matching rows from both tables. However, a full outer join also returns non-matching rows from both tables; left and right. While DB2 UDB for iSeries does not support full outer join syntax, you can simulate a full outer join by using a left outer join and a right exception join. Suppose you want to find all employees and all projects. You want to see those employees that are not currently in charge of a project as well. The following query will return a list of all employees whose names are greater than 'S', along with their assigned project numbers.

```
SELECT EMPNO, LASTNAME, PROJNO
   FROM CORPDATA.EMPLOYEE LEFT OUTER JOIN CORPDATA.PROJECT
      ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S'
 UNION
 (SELECT EMPNO, LASTNAME, PROJNO
   FROM CORPDATA.PROJECT EXCEPTION JOIN CORPDATA.EMPLOYEE
      ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S');
```

Multiple join types in one statement

There are times when more than two tables need to be joined to produce the desired result. If you wanted to return all the employees, their department name, and the project they are responsible for, if any, the EMPLOYEE table,

DEPARTMENT table, and PROJECT table would all need to be joined to get the information. The following example shows the query and the results.

```

SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
FROM CORPDATA.EMPLOYEE INNER JOIN CORPDATA.DEPARTMENT
    ON WORKDEPT = DEPTNO
LEFT OUTER JOIN CORPDATA.PROJECT
    ON EMPNO = RESPEMP
WHERE LASTNAME > 'S'

```

The result of this query is:

EMPNO	LASTNAME	DEPTNAME	PROJNO
000020	THOMPSON	PLANNING	PL2100
000060	STERN	MANUFACTURING SYSTEMS	MA2110
000100	SPENSER	SOFTWARE SUPPORT	OP2010
000170	YOSHIMURA	MANUFACTURING SYSTEMS	-
000180	SCOUTTEN	MANUFACTURING SYSTEMS	-
000190	WALKER	MANUFACTURING SYSTEMS	-
000250	SMITH	ADMINISTRATION SYSTEMS	AD3112
000280	SCHNEIDER	OPERATIONS	-
000300	SMITH	OPERATIONS	-
000310	SETRIGHT	OPERATIONS	-

For more information about joins, see the SQL Reference book.

Using table expressions

You can use table expressions to specify an intermediate result table. Table expressions can be used in place of a view to avoid creating the view when general use of the view is not required. Table expressions consist of nested table expressions (also called derived tables) and common table expressions.

Nested table expressions are specified within parentheses in the FROM clause. For example, suppose you want a result table that shows the manager number, department number, and maximum salary for each department. The manager number is in the DEPARTMENT table, the department number is in both the DEPARTMENT and EMPLOYEE tables, and the salaries are in the EMPLOYEE table. You can use a table expression in the FROM clause to select the maximum salary for each department. You can also add a correlation name, T2, following the nested table expression to name the derived table. The outer select then uses T2 to qualify columns that are selected from the derived table, in this case MAXSAL and WORKDEPT. Note that the MAX(SALARY) column selected in the nested table expression must be named in order to be referenced in the outer select. The AS clause is used to do that.

```

SELECT MGRNO, T1.DEPTNO, MAXSAL
FROM CORPDATA.DEPARTMENT T1,
    (SELECT MAX(SALARY) AS MAXSAL, WORKDEPT
     FROM CORPDATA.EMPLOYEE E1
     GROUP BY WORKDEPT) T2
WHERE T1.DEPTNO = T2.WORKDEPT
ORDER BY DEPTNO

```

The result of the query is:

MGRNO	DEPTNO	MAXSAL
000010	A00	52750.00
000020	B01	41250.00
000030	C01	38250.00
000060	D11	32250.00
000070	D21	36170.00
000050	E01	40175.00
000090	E11	29750.00
000100	E21	26150.00

Common table expressions can be specified prior to the full-select in a SELECT statement, an INSERT statement, or a CREATE VIEW statement. They can be used when the same result table needs to be shared in a full-select. Common table expressions are preceded with the keyword WITH.

For example, suppose you want a table that shows the minimum and maximum of the average salary of a certain set of departments. The first character of the department number has some meaning and you want to get the minimum and maximum for those departments that start with the letter 'D' and those that start with the letter 'E'. You can use a common table expression to select the average salary for each department. Again, you must name the derived table; in this case, the name is DT. You can then specify a SELECT statement using a WHERE clause to restrict the selection to only the departments that begin with a certain letter. Specify the minimum and maximum of column AVGSAL from the derived table DT. Specify a UNION to get the results for the letter 'E' and the results for the letter 'D'.

```
WITH DT AS (SELECT E.WORKDEPT AS DEPTNO, AVG(SALARY) AS AVGSAL
            FROM CORPDATA.DEPARTMENT D , CORPDATA.EMPLOYEE E
            WHERE D.DEPTNO = E.WORKDEPT
            GROUP BY E.WORKDEPT)
SELECT 'E', MAX(AVGSAL), MIN(AVGSAL) FROM DT
WHERE DEPTNO LIKE 'E%'
UNION
SELECT 'D', MAX(AVGSAL), MIN(AVGSAL) FROM DT
WHERE DEPTNO LIKE 'D%'
```

The result of the query is:

	MAX(AVGSAL)	MIN(AVGSAL)
E	40175.00	21020.00
D	25668.57	25147.27

Suppose you want to write a query against your ordering database that will return the top 5 items (in total quantity ordered) within the last 1000 orders from customers who also ordered item 'XXX'.

```
WITH X AS (SELECT ORDER_ID, CUST_ID
            FROM ORDERS
            ORDER BY ORD_DATE DESC
            FETCH FIRST 1000 ROWS ONLY),
Y AS (SELECT CUST_ID, LINE_ID, ORDER_QTY
        FROM X, ORDERLINE
```

```

WHERE X.ORDER_ID = ORDERLINE.ORDER_ID)
SELECT LINE_ID
FROM (SELECT LINE_ID
FROM Y
WHERE Y.CUST_ID IN (SELECT DISTINCT CUST_ID
FROM Y
WHERE LINE.ID = 'XXX' )
GROUP BY LINE_ID
ORDER BY SUM(ORDER_QTY) DESC)
FETCH FIRST 5 ROWS ONLY

```

The first common table expression (X) returns the most recent 1000 order numbers. The result is ordered by the date in descending order and then only the first 1000 of those ordered rows are returned as the result table.

The second common table expression (Y) joins the most recent 1000 orders with the line item table and returns (for each of the 1000 orders) the customer, line item, and quantity of the line item for that order.

The derived table in the main select statement returns the line items for the customers who are in the top 1000 orders who ordered item XXX. The results for all customers who ordered XXX are then grouped by the line item and the groups are ordered by the total quantity of the line item.

Finally, the outer select selects only the first 5 rows from the ordered list that the derived table returned.

Using the UNION keyword to combine subselects

Using the UNION keyword, you can combine two or more subselects to form a fullselect. When SQL encounters the UNION keyword, it processes each subselect to form an interim result table, then it combines the interim result table of each subselect and deletes duplicate rows to form a combined result table. You use UNION to merge lists of values from two or more tables. You can use any of the clauses and techniques you have learned so far when coding select-statements.

You can use UNION to eliminate duplicates when merging lists of values obtained from several tables. For example, you can obtain a combined list of employee numbers that includes:

- People in department D11
- People whose assignments include projects MA2112, MA2113, and AD3111

The combined list is derived from two tables and contains no duplicates. To do this, specify:

```

SELECT EMPNO
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D11'
UNION
SELECT EMPNO
FROM CORPDATA.EMPPROJECT
WHERE PROJNO = 'MA2112' OR
PROJNO = 'MA2113' OR
PROJNO = 'AD3111'
ORDER BY EMPNO

```

You can also use UNION in a common table expression, nested table expression, or when creating a view. See “Creating a view with UNION” on page 57 for details.

To better understand the results from these SQL statements, imagine that SQL goes through the following process:

Step 1. SQL processes the first SELECT statement:

```
SELECT EMPNO
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D11'
```

Which results in an interim result table:

EMPNO from CORPDATA.EMPLOYEE
000060
000150
000160
000170
000180
000190
000200
000210
000220
200170
200220

Step 2. SQL processes the second SELECT statement:

```
SELECT EMPNO
FROM CORPDATA.EMPPROJACT
WHERE PROJNO= 'MA2112' OR
        PROJNO= 'MA2113' OR
        PROJNO= 'AD3111'
```

Which results in another interim result table:

EMPNO from CORPDATA.EMPPROJACT
000230
000230
000240
000230
000230
000240
000230
000150
000170
000190
000170
000190
000150
000160
000180

EMPNO from CORPDATA.EMPPROJACT

000170

000210

000210

Step 3. SQL combines the two interim result tables, removes duplicate rows, and orders the result:

```
SELECT EMPNO
      FROM CORPDATA.EMPLOYEE
      WHERE WORKDEPT = 'D11'
UNION
SELECT EMPNO
      FROM CORPDATA.EMPPROJACT
      WHERE PROJNO='MA2112' OR
             PROJNO= 'MA2113' OR
             PROJNO= 'AD3111'
ORDER BY EMPNO
```

Which results in a combined result table with values in ascending sequence:

EMPNO

000060

000150

000160

000170

000180

000190

000200

000210

000220

000230

000240

200170

200220

When you use UNION:

- Any ORDER BY clause must appear after the last subselect that is part of the union. In this example, the results are sequenced on the basis of the first selected column, *EMPNO*. The ORDER BY clause specifies that the combined result table is to be in collated sequence. ORDER BY is not allowed in a view.
- A name may be specified on the ORDER BY clause if the result columns are named. A result column is named if the corresponding columns in each of the unioned select-statements have the same name. An AS clause can be used to assign a name to columns in the select list.

```
SELECT A + B AS X ...
UNION
SELECT X ... ORDER BY X
```

If the result columns are unnamed, use a positive integer to order the result. The number refers to the position of the expression in the list of expressions you include in your subselects.

```

SELECT A + B ...
UNION
SELECT X ... ORDER BY 1

```

To identify which subselect each row is from, you can include a constant at the end of the select list of each subselect in the union. When SQL returns your results, the last column contains the constant for the subselect that is the source of that row. For example, you can specify:

```

SELECT A, B, 'A1' ...
UNION
SELECT X, Y, 'B2'...

```

When a row is returned, it includes a value (either A1 or B2) to indicate the table that is the source of the row's values. If the column names in the union are different, SQL uses the set of column names specified in the first subselect when interactive SQL displays or prints the results, or in the SQLDA resulting from processing an SQL DESCRIBE statement.

For information on compatibility of the length and data type for columns in a UNION, see the Rules for result data type topic in the SQL Reference book.

Note: Sort sequence is applied after the fields across the UNION pieces are made compatible. The sort sequence is used for the distinct processing that implicitly occurs during UNION processing. See Chapter 8, "Sort sequences in SQL" on page 115 for more details about sort sequence.

Specifying UNION ALL

If you want to keep duplicates in the result of a UNION, specify UNION ALL instead of just UNION. Using the same as steps and example as UNION:

Step 3. SQL combines two interim result tables:

```

SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'D11'
UNION ALL
SELECT EMPNO
  FROM CORPDATA.EMPPROJACT
  WHERE PROJNO='MA2112' OR
        PROJNO= 'MA2113' OR
        PROJNO= 'AD3111'
ORDER BY EMPNO

```

Resulting in an ordered result table that includes duplicates:

EMPNO
000060
000150
000150
000150
000160
000160
000170
000170
000170

EMPNO

000170

000180

000180

000190

000190

000190

000200

000210

000210

000210

000220

000230

000230

000230

000230

000230

000240

000240

200170

200220

The UNION ALL operation is associative, for example:

```
(SELECT PROJNO FROM CORPDATA.PROJECT
UNION ALL
SELECT PROJNO FROM CORPDATA.PROJECT)
UNION ALL
SELECT PROJNO FROM CORPDATA.EMPPROJECT
```

This statement can also be written as:

```
SELECT PROJNO FROM CORPDATA.PROJECT
UNION ALL
(SELECT PROJNO FROM CORPDATA.PROJECT
UNION ALL
SELECT PROJNO FROM CORPDATA.EMPPROJECT)
```

When you include the UNION ALL in the same SQL statement as a UNION operator, however, the result of the operation depends on the order of evaluation. Where there are no parentheses, evaluation is from left to right. Where parentheses are included, the parenthesized subselect is evaluated first, followed, from left to right, by the other parts of the statement.

Data retrieval errors

If SQL finds that a retrieved character or graphic column is too long to be placed in a host variable, SQL does the following:

- Truncates the data while assigning the value to the host variable.
- Sets SQLWARN0 and SQLWARN1 in the SQLCA to the value 'W'.

- Sets the indicator variable, if provided, to the length of the value before truncation.

If SQL finds a data mapping error while running a statement, one of two things occurs:

- If the error occurs on an expression in the SELECT list and an indicator variable is provided for the expression in error:
 - SQL returns a –2 for the indicator variable corresponding to the expression in error.
 - SQL returns all valid data for that row.
 - SQL returns a positive SQLCODE.
- If an indicator variable is not provided, SQL returns the corresponding negative SQLCODE in the SQLCA.

Data mapping errors include:

- +138 - Argument of the substrings function is not valid.
- +180 - Syntax for a string representation of a date, time, or timestamp is not valid.
- +181 - String representation of a date, time, or timestamp is not a valid value.
- +183 - Invalid result from a date/time expression. The resulting date or timestamp is not within the valid range of dates or timestamps.
- +191 - MIXED data is not properly formed.
- +304 - Numeric conversion error (for example, overflow, underflow, or division by zero).
- +331 - Characters cannot be converted.
- +420 - Character in the CAST argument is not valid.
- +802 - Data conversion or data mapping error.

For data mapping errors, the SQLCA reports only the last error detected. The indicator variable corresponding to each result column having an error is set to –2.

If the full-select contains DISTINCT in the select list and a column in the select list contains numeric data that is not valid, the data is considered equal to a null value if the query is completed as a sort. If an existing index is used, the data is not considered equal to a null.

The impact of data mapping errors on the ORDER BY clause depends on the situation:

- If the data mapping error occurs while data is being assigned to a host variable in a SELECT INTO or FETCH statement, and that same expression is used in the ORDER BY clause, the result record is ordered based on the value of the expression. It is not ordered as if it were a null (higher than all other values). This is because the expression was evaluated before the assignment to the host variable is attempted.
- If the data mapping error occurs while an expression in the select-list is being evaluated and the same expression is used in the ORDER BY clause, the result column is normally ordered as if it were a null value (higher than all other values). If the ORDER BY clause is implemented by using a sort, the result column is ordered as if it were a null value. If the ORDER BY clause is implemented by using an existing index, in the following cases, the result column is ordered based on the actual value of the expression in the index:

- The expression is a date column with a date format of *MDY, *DMY, *YMD, or *JUL, and a date conversion error occurs because the date is not within the valid range for dates.
- The expression is a character column and a character could not be converted.
- The expression is a decimal column and a numeric value that is not valid is detected.

Chapter 6. SQL Insert, Update, and Delete

This section shows the basic SQL statements and clauses that update, delete, and insert data into tables and views. The SQL statements used are UPDATE, DELETE, and INSERT. Examples using these SQL statements are supplied to help you develop SQL applications. Detailed syntax and parameter descriptions for SQL statements are given in the SQL Reference book.

Notes:

1. The SQL statements described in this section can be run on SQL tables and views, and database physical and logical files. The tables, views, and files can be either in a schema or in a library.
2. Character strings specified in an SQL statement (such as those used with WHERE or VALUES clauses) are case sensitive; that is, uppercase characters must be entered in uppercase and lowercase characters must be entered in lowercase.

WHERE ADMRDEPT='a00' (does not return a result)

WHERE ADMRDEPT='A00' (returns a valid department number)

A character string may not be case sensitive if a shared-weight sort sequence is being used where uppercase and lowercase characters are treated as the same character. See Chapter 8, “Sort sequences in SQL” on page 115 for more information about sort sequence.

The sections included in this chapter are:

- “Inserting rows using the INSERT statement”
- “Changing data in a table using the UPDATE statement” on page 97
- “Removing rows from a table using the DELETE statement” on page 102

Inserting rows using the INSERT statement

You can use the INSERT statement to add new rows to a table or view in one of the following ways:

- Specifying values in the INSERT statement for columns of the single row to be added.
- Including a select-statement in the INSERT statement to tell SQL what data for the new row is contained in another table or view. “Inserting rows into a table using a Select-Statement” on page 95 explains how to use the select-statement within an INSERT statement to add zero, one, or many rows to a table.
- Specifying the blocked form of the INSERT statement to add multiple rows. “Inserting multiple rows in a table with the blocked INSERT statement” on page 96 explains how to use the blocked form of the INSERT statement to add multiple rows to a table.

Note: Because views are built on tables and actually contain no data, working with views can be confusing. See “Creating and using views” on page 55 for more information and restrictions about inserting data by using a view.

For a complete description of INSERT, see INSERT statement in the SQL Reference.

For every row you insert, you must supply a value for each column defined with the NOT NULL attribute if that column does not have a default value. The INSERT statement for adding a row to a table or view may look like this:

```
INSERT INTO table-name
(column1, column2, ... )
VALUES (value-for-column1, value-for-column2, ... )
```

The INTO clause names the columns for which you specify values. The VALUES clause specifies a value for each column named in the INTO clause. The value you specify can be:

A **constant**. Inserts the value provided in the VALUES clause.

A **null value**. Inserts the null value, using the keyword NULL. The column must be defined as capable of containing a null value or an error occurs.

A **host variable**. Inserts the contents of a host variable.

A **special register**. Inserts a special register value; for example, USER.

An **expression**. Inserts the value that results from an expression.

A **subquery** inserts the value that is the result of running the select statement.

The **DEFAULT** keyword. Inserts the default value of the column. The column must have a default value defined for it or allow the NULL value, or an error occurs.

You must provide a value in the VALUES clause for each column named in an INSERT statement's column list. The column name list can be omitted if all columns in the table have a value provided in the VALUES clause. If a column has a default value, the keyword DEFAULT may be used as a value in the VALUES clause. This causes the default value for the column to be placed in the column.

It is a good idea to name all columns into which you are inserting values because:

- Your INSERT statement is more descriptive.
- You can verify that you are providing the values in the proper order based on the column names.
- You have better data independence. The order in which the columns are defined in the table does not affect your INSERT statement.

If the column is defined to allow null values or to have a default, you do not need to name it in the column name list or specify a value for it. The default value is used. If the column is defined to have a default value, the default value is placed in the column. If DEFAULT was specified for the column definition without an explicit default value, SQL places the default value for that data type in the column. If the column does not have a default value defined for it, but is defined to allow the null value (NOT NULL was not specified in the column definition), SQL places the null value in the column.

- For numeric columns, the default value is 0.
- For fixed length character or graphic columns, the default is blanks.
- For varying length character or graphic columns or LOB columns, the default is a zero length string.
- For date, time, and timestamp columns, the default value is the current date, time, or timestamp. When inserting a block of records, the default date/time value is extracted from the system when the block is written. This means that the column will be assigned the same default value for each row in the block.
- For DataLink columns, the default value corresponds to DLVALUE('','URL',').

- For distinct-type columns, the default value is the default value of the corresponding source type.
- For ROWID columns or columns that are defined AS IDENTITY, the database manager generates a default value. See “Inserting into an identity column” on page 97.

When your program attempts to insert a row that duplicates another row already in the table, an error might occur. Multiple null values may or may not be considered duplicate values, depending on the option used when the index was created.

- If the table has a primary key, unique key, or unique index, the row is not inserted. Instead, SQL returns an SQLCODE of -803.
- If the table does not have a primary key, unique key, or unique index, the row can be inserted without error.

If SQL finds an error while running the INSERT statement, it stops inserting data. If you specify COMMIT(*ALL), COMMIT(*CS), COMMIT(*CHG), or COMMIT(*RR), no rows are inserted. Rows already inserted by this statement, in the case of INSERT with a select-statement or blocked insert, are deleted. If you specify COMMIT(*NONE), any rows already inserted are *not* deleted.

A table created by SQL is created with the Reuse Deleted Records parameter of *YES. This allows the database manager to reuse any rows in the table that were marked as deleted. The CHGPF command can be used to change the attribute to *NO. This causes INSERT to always add rows to the end of the table.

The order in which rows are inserted does not guarantee the order in which they will be retrieved.

If the row is inserted without error, the SQLERRD(3) field of the SQLCA has a value of 1.

Note: For blocked INSERT or for INSERT with select-statement, more than one row can be inserted. The number of rows inserted is reflected in SQLERRD(3).

Inserting rows into a table using a Select-Statement

You can use a select-statement within an INSERT statement to insert zero, one, or more rows selected from the table or view you specify into another table. The table you select the rows from can be the same table you are inserting into. If they are the same table, SQL will create a temporary result table containing the selected rows and then insert from the temporary table into the target table.

One use for this kind of INSERT statement is to move data into a table you created for summary data. For example, suppose you want a table that shows each employee’s time commitments to projects. You could create a table called EMPTIME with the columns EMPNUMBER, PROJNUMBER, STARTDATE, and ENDDATE and then use the following INSERT statement to fill the table:

```
INSERT INTO CORPDATA.EMPTIME
  (EMPNUMBER, PROJNUMBER, STARTDATE, ENDDATE)
SELECT EMPNO, PROJNO, EMSTDATE, EMENDATE
FROM CORPDATA.EMPPROJECT
```

The select-statement embedded in the INSERT statement is no different from the select-statement you use to retrieve data. With the exception of FOR READ ONLY,

FOR UPDATE, or the OPTIMIZE clause, you can use all the keywords, column functions, and techniques used to retrieve data. SQL inserts all the rows that meet the search conditions into the table you specify. Inserting rows from one table into another table does not affect any existing rows in either the source table or the target table.

Notes on multiple-row insertion

You should consider the following when inserting multiple rows into a table:

- The number of columns implicitly or explicitly listed in the INSERT statement must equal the number of columns listed in the select-statement.
- The data in the columns you are selecting must be compatible with the columns you are inserting into when using the INSERT with select-statement.
- In the event the select-statement embedded in the INSERT returns no rows, an SQLCODE of 100 is returned to alert you that no rows were inserted. If you successfully insert rows, the SQLERRD(3) field of the SQLCA has an integer representing the number of rows SQL actually inserted.
- If SQL finds an error while running the INSERT statement, SQL stops the operation. If you specify COMMIT (*CHG), COMMIT(*CS), COMMIT (*ALL), or COMMIT(*RR), nothing is inserted into the table and a negative SQLCODE is returned. If you specify COMMIT(*NONE), any rows inserted prior to the error remain in the table.
- You can join two or more tables with a select-statement in an INSERT statement. Loaded in this manner, the table can be operated on with UPDATE, DELETE, and INSERT statements, because the rows exist as physically stored rows in a table.

Inserting multiple rows in a table with the blocked INSERT statement

A blocked INSERT can be used to insert multiple rows into a table with a single statement. The blocked INSERT statement is supported in all of the languages except REXX. The data inserted into the table must be in a host structure array. If indicator variables are used with a blocked INSERT, they must also be in a host structure array. For information on host structure arrays for a particular language, refer to the chapter on that language in the SQL Programming with Host Languages book.

For example, to add ten employees to the CORPDATA.EMPLOYEE table:

```
INSERT INTO CORPDATA.EMPLOYEE  
            (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT)  
10 ROWS VALUES (:DSTRUCT:ISTRUCT)
```

DSTRUCT is a host structure array with five elements that is declared in the program. The five elements correspond to EMPNO, FIRSTNME, MIDINIT, LASTNAME, and WORKDEPT. DSTRUCT has a dimension of at least ten to accommodate inserting ten rows. ISTRUCT is a host structure array that is declared in the program. ISTRUCT has a dimension of at least ten small integer fields for the indicators.

Blocked INSERT statements are supported for non-distributed SQL applications and for distributed applications where both the application server and the application requester are iSeries systems.

Inserting into an identity column

You can insert a value into an identity column or allow the system to insert a value for you. For example, the table created in “Creating and altering an identity column” on page 50, has columns called ORDERNO (identity column), SHIPPED_TO (varchar(36)), and ORDER_DATE (date). You can insert a row into this table by issuing the following statement:

```
INSERT INTO ORDERS (SHIPPED_TO, ORDER_DATE)
VALUES ('BME TOOL', 2002-02-04)
```

In this case, a value is generated by the system for the identity column automatically. You could also write this statement using the DEFAULT keyword:

```
INSERT INTO ORDERS (SHIPPED_TO, ORDER_DATE, ORDERNO)
VALUES ('BME TOOL', 2002-02-04, DEFAULT)
```

After the insert, you can use the IDENTITY_VAL_LOCAL function to determine the value that the system assigned to the column. See IDENTITY_VAL_LOCAL function in the SQL Reference for more details and examples.

Sometimes a value for an identity column is specified by the user, such as in this INSERT statement using a SELECT:

```
INSERT INTO ORDERS OVERRIDING USER VALUE
(SELECT * FROM TODAYS_ORDER)
```

In this case, OVERRIDING USER VALUE tells the system to ignore the value provided for the identity column from the SELECT and to generate a new value for the identity column. OVERRIDING USER VALUE must be used if the identity column was created with the GENERATED ALWAYS clause; it is optional for GENERATED BY DEFAULT. If OVERRIDING USER VALUE is not specified for a GENERATED BY DEFAULT identity column, the value provided for the column in the SELECT is inserted.

You can force the system to use the value from the select for a GENERATED ALWAYS identity column by specifying OVERRIDING SYSTEM VALUE. For example, issue the following statement:

```
INSERT INTO ORDERS OVERRIDING SYSTEM VALUE
(SELECT * FROM TODAYS_ORDER)
```

This INSERT statement uses the value from SELECT; it does not generate a new value for the identity column. You cannot provide a value for an identity column created using GENERATED ALWAYS without using the OVERRIDING SYSTEM VALUE clause.

Changing data in a table using the UPDATE statement

To change the data in a table, use the UPDATE statement. With the UPDATE statement, you can change the value of one or more columns in each row that satisfies the search condition of the WHERE clause. The result of the UPDATE statement is one or more changed column values in zero or more rows of a table (depending on how many rows satisfy the search condition specified in the WHERE clause). The UPDATE statement looks like this:

```
UPDATE table-name
SET column-1 = value-1,
    column-2 = value-2, ...
WHERE search-condition ...
```

For example, suppose an employee was relocated. To update several items of the employee's data in the CORPDATA.EMPLOYEE table to reflect the move, you can specify:

```
UPDATE CORPDATA.EMPLOYEE
SET JOB = :PGM-CODE,
    PHONENO = :PGM-PHONE
WHERE EMPNO = :PGM-SERIAL
```

Use the SET clause to specify a new value for each column you want to update. The SET clause names the columns you want updated and provides the values you want them changed to. The value you specify can be:

A **column name**. Replace the column's current value with the contents of another column in the same row.

A **constant**. Replace the column's current value with the value provided in the SET clause.

A **null value**. Replace the column's current value with the null value, using the keyword NULL. The column must be defined as capable of containing a null value when the table was created, or an error occurs.

A **host variable**. Replace the column's current value with the contents of a host variable.

A **special register**. Replace the column's current value with a special register value; for example, USER.

An **expression**. Replace the column's current value with the value that results from an expression.

A **scalar subselect**. Replace the column's current value with the value that the subquery returns.

The **DEFAULT** keyword. Replace the column's current value with the default value of the column. The column must have a default value defined for it or allow the NULL value, or an error occurs.

The following is an example of a statement that uses many different values:

```
UPDATE WORKTABLE
SET COL1 = 'ASC',
    COL2 = NULL,
    COL3 = :FIELD3,
    COL4 = CURRENT TIME,
    COL5 = AMT - 6.00,
    COL6 = COL7
WHERE EMPNO = :PGM-SERIAL
```

To identify the rows to be updated, use the WHERE clause:

- To update a single row, use a WHERE clause that selects only one row.
- To update several rows, use a WHERE clause that selects only the rows you want to update.

You can omit the WHERE clause. If you do, SQL updates each row in the table or view with the values you supply.

If the database manager finds an error while running your UPDATE statement, it stops updating and returns a negative SQLCODE. If you specify COMMIT(*ALL), COMMIT(*CS), COMMIT(*CHG), or COMMIT(*RR), no rows in the table are changed (rows already changed by this statement, if any, are restored to their previous values). If COMMIT(*NONE) is specified, any rows already changed are *not* restored to previous values.

If the database manager cannot find any rows that satisfy the search condition, an SQLCODE of +100 is returned.

Note: The UPDATE statement may have updated more than one row. The number of rows updated is reflected in SQLERRD(3) of the SQLCA.

The SET clause of an UPDATE statement can be used in many ways to determine the actual values to be set in each row being updated. The following example lists each column with its corresponding value:

```
UPDATE EMPLOYEE
SET  WORKDEPT = 'D11',
     PHONENO  = '7213',
     JOB      = 'DESIGNER'
WHERE EMPNO  = '000270'
```

The previous update can also be written by specifying all of the columns and then all of the values:

```
UPDATE EMPLOYEE
SET  (WORKDEPT, PHONENO, JOB)
     = ('D11', '7213', 'DESIGNER')
WHERE EMPNO = '000270'
```

For more ways of updating data in a table, see the following sections:

- “Updating a table using a scalar-subselect”
- “Updating a table with rows from another table”
- “Updating data as it is retrieved from a table” on page 100

For a complete description of the UPDATE statement, see UPDATE in the SQL Reference.

Updating a table using a scalar-subselect

Another way to select a value (or multiple values) for an update is to use a scalar-subselect. The scalar-subselect allows you to update one or more columns by setting them to one or more values selected from another table. In the following example, an employee moves to a different department but continues working on the same projects. The employee table has already been updated to contain the new department number. Now the project table needs to be updated to reflect the new department number of this employee (employee number is '000030').

```
UPDATE PROJECT
SET  DEPTNO =
     (SELECT WORKDEPT FROM EMPLOYEE
      WHERE PROJECT.RESPEMP = EMPLOYEE.EMPNO)
WHERE RESPEMP='000030'
```

This same technique could be used to update a list of columns with multiple values returned from a single select.

Updating a table with rows from another table

It is also possible to update an entire row in one table with values from a row in another table. Suppose there is a master class schedule table that needs to be updated with changes that have been made in a copy of the table. The changes are made to the work copy and merged into the master table every night. The two tables have exactly the same columns and one column, CLASS_CODE, is a unique key column.

```

UPDATE CL_SCHED
  SET ROW_ =
    (SELECT * FROM MYCOPY
     WHERE CL_SCHED.CLASS_CODE = MYCOPY.CLASS_CODE)

```

This update will update all of the rows in CL_SCHED with the values from MYCOPY.

Updating an identity column

You can update the value in an identity column to a specified value or have the system generate a new value. For example, using the table created in “Creating and altering an identity column” on page 50, with columns called ORDERNO (identity column), SHIPPED_TO (varchar(36)), and ORDER_DATE (date), you can update the value in an identity column by issuing the following statement:

```

UPDATE ORDERS
  SET (ORDERNO, ORDER_DATE)=
    (DEFAULT, 2002-02-05)
  WHERE SHIPPED_TO = 'BME TOOL'

```

A value is generated by the system for the identity column automatically. You can override having the system generate a value by using the OVERRIDING SYSTEM VALUE clause:

```

UPDATE ORDERS OVERRIDING SYSTEM VALUE
  SET (ORDERNO, ORDER_DATE)=
    (553, '2002-02-05')
  WHERE SHIPPED_TO = 'BME TOOL'

```

Updating data as it is retrieved from a table

You can update rows of data as you retrieve them by using a cursor. See Chapter 9, “Using a Cursor” on page 121 for more information about cursors. On the select-statement, use FOR UPDATE OF followed by a list of columns that may be updated. Then use the cursor-controlled UPDATE statement. The WHERE CURRENT OF clause names the cursor that points to the row you want to update. If a FOR UPDATE OF, an ORDER BY, a FOR READ ONLY, or a SCROLL clause without the DYNAMIC clause is not specified, all columns can be updated.

If a multiple-row FETCH statement has been specified and run, the cursor is positioned on the last row of the block. Therefore, if the WHERE CURRENT OF clause is specified on the UPDATE statement, the last row in the block is updated. If a row within the block must be updated, the program must first position the cursor on that row. Then the UPDATE WHERE CURRENT OF can be specified. Consider the following example:

Table 8. Updating a Table

Scrollable Cursor SQL Statement	Comments
<pre> EXEC SQL DECLARE THISEMP DYNAMIC SCROLL CURSOR FOR SELECT EMPNO, WORKDEPT, BONUS FROM CORPDATA.EMPLOYEE WHERE WORKDEPT = 'D11' FOR UPDATE OF BONUS END-EXEC. </pre>	
<pre> EXEC SQL OPEN THISEMP END-EXEC. </pre>	

Table 8. Updating a Table (continued)

Scrollable Cursor SQL Statement	Comments
EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.	
EXEC SQL FETCH NEXT FROM THISEMP FOR 5 ROWS INTO :DEPTINFO :IND-ARRAY END-EXEC.	DEPTINFO and IND-ARRAY are declared in the program as a host structure array and an indicator array.
... determine if any employees in department D11 receive a bonus less than \$500.00. If so, update that record to the new minimum of \$500.00.	
EXEC SQL FETCH RELATIVE :NUMBACK FROM THISEMP END-EXEC.	... positions to the record in the block to update by fetching in the reverse order.
EXEC SQL UPDATE CORPDATA.EMPLOYEE SET BONUS = 500 WHERE CURRENT OF THISEMP END-EXEC.	... updates the bonus for the employee in department D11 that is under the new \$500.00 minimum.
EXEC SQL FETCH RELATIVE :NUMBACK FROM THISEMP FOR 5 ROWS INTO :DEPTINFO :IND-ARRAY END-EXEC.	... positions to the beginning of the same block that was already fetched and fetches the block again. (NUMBACK - (5 - NUMBACK - 1))
... branch back to determine if any more employees in the block have a bonus under \$500.00.	
... branch back to fetch and process the next block of rows.	
CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.	

Restrictions

You cannot use FOR UPDATE OF with a select-statement that includes any of these elements:

- The first FROM clause identifies more than one table or view.
- The first FROM clause identifies a read-only view.
- The first SELECT clause specifies the keyword DISTINCT.
- The first FROM clause identifies a user-defined table function.
- The outer subselect contains a GROUP BY clause.
- The outer subselect contains a HAVING clause.
- The first SELECT clause contains a column function.
- The select-statement contains a UNION or UNION ALL operator.
- The select-statement contains an ORDER BY clause, and the FOR UPDATE OF clause and DYNAMIC SCROLL are not specified.
- The select-statement includes a FOR FETCH ONLY clause.

- The SCROLL keyword is specified without DYNAMIC.
- The select list includes a DATALINK column and a FOR UPDATE OF clause is not specified.
- The first subselect requires a temporary result table.
- The select-statement includes a FETCH FIRST *n* ROWS ONLY.

If a FOR UPDATE OF clause is specified, you cannot update columns that were not named in the FOR UPDATE OF clause. But you can name columns in the FOR UPDATE OF clause that are not in the SELECT list, as in this example:

```
SELECT A, B, C FROM TABLE
FOR UPDATE OF A,E
```

Do not name more columns than you need in the FOR UPDATE OF clause; indexes on those columns are not used when you access the table.

Removing rows from a table using the DELETE statement

To remove rows from a table, use the DELETE statement. When you DELETE a row, you remove the entire row. DELETE does not remove specific columns from the row. The result of the DELETE statement is the removal of zero or more rows of a table (depending on how many rows satisfy the search condition specified in the WHERE clause). If you omit the WHERE clause from a DELETE statement, SQL removes all the rows of the table. The DELETE statement looks like this:

```
DELETE FROM table-name
WHERE search-condition ...
```

For example, suppose department D11 was moved to another place. You want to delete each row in the CORPDATA.EMPLOYEE table with a WORKDEPT value of D11 as follows:

```
DELETE FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D11'
```

The WHERE clause tells SQL which rows you want to delete from the table. SQL deletes all the rows that satisfy the search condition from the base table. Deleting rows from a view deletes the rows from the base table. You can omit the WHERE clause, but it is best to include one, because a DELETE statement without a WHERE clause deletes all the rows from the table or view. To delete a table definition as well as the table contents, issue the DROP statement. For more information about the DROP statement, see the DROP statement topic in the *SQL Reference* book.

If SQL finds an error while running your DELETE statement, it stops deleting data and returns a negative SQLCODE. If you specify COMMIT(*ALL), COMMIT(*CS), COMMIT(*CHG), or COMMIT(*RR), no rows in the table are deleted (rows already deleted by this statement, if any, are restored to their previous values). If COMMIT(*NONE) is specified, any rows already deleted are *not* restored to their previous values.

If SQL cannot find any rows that satisfy the search condition, an SQLCODE of +100 is returned.

Note: The DELETE statement may have deleted more than one row. The number of rows deleted is reflected in SQLERRD(3).

For more information about the DELETE statement, see the DELETE statement topic in the *SQL Reference* book.

Chapter 7. Using Subqueries

You can use subqueries in a search condition as another way to select your data. Subqueries can be used in expressions, in the select-list, and in the ORDER BY and the GROUP BY clauses. For more details, see the following sections:

- “Subqueries in SELECT statements”
- “Notes on using subqueries” on page 108
- “Correlated subqueries” on page 109
- “Using correlated subqueries in an UPDATE statement” on page 112
- “Using correlated subqueries in a DELETE statement” on page 113

Subqueries in SELECT statements

In simple WHERE and HAVING clauses, you can specify a search condition by using a literal value, a column name, an expression, or a special register. In those search conditions, you know that you are searching for a specific value. However, sometimes you cannot supply that value until you have retrieved other data from a table. For example, suppose you want a list of the employee numbers, names, and job codes of all employees working on a particular project, say project number MA2100. The first part of the statement is easy to write:

```
SELECT EMPNO, LASTNAME, JOB
FROM CORPDATA.EMPLOYEE
WHERE EMPNO ...
```

But you cannot go further because the CORPDATA.EMPLOYEE table does not include project number data. You do not know which employees are working on project MA2100 without issuing another SELECT statement against the CORPDATA.EMP_ACT table.

With SQL, you can nest one SELECT statement within another to solve this problem. The inner SELECT statement is called a **subquery**. The SELECT statement surrounding the subquery is called the **outer-level SELECT**. Using a subquery, you could issue just one SQL statement to retrieve the employee numbers, names, and job codes for employees who work on project MA2100:

```
SELECT EMPNO, LASTNAME, JOB
FROM CORPDATA.EMPLOYEE
WHERE EMPNO IN
  (SELECT EMPNO
   FROM CORPDATA.EMPPROJACT
   WHERE PROJNO = 'MA2100')
```

To better understand what will result from this SQL statement, imagine that SQL goes through the following process:

Step 1: SQL evaluates the subquery to obtain a list of EMPNO values:

```
(SELECT EMPNO
 FROM CORPDATA.EMPPROJACT
 WHERE PROJNO= 'MA2100')
```

Which results in an interim results table:

EMPNO from CORPDATA.EMPPROJACT
000010
000110

Step 2: The interim results table then serves as a list in the search condition of the outer-level SELECT. Essentially, this is the statement that is run.

```
SELECT EMPNO, LASTNAME, JOB
FROM CORPDATA.EMPLOYEE
WHERE EMPNO IN
      ('000010', '000110')
```

The final result table looks like this:

EMPNO	LASTNAME	JOB
000010	HAAS	PRES
000110	LUCCHESI	SALESREP

For more details, see the following sections:

- “Correlation”
- “Subqueries and search conditions”
- “How subqueries are used” on page 107
- “Notes on using subqueries” on page 108
- “Correlated subqueries” on page 109
- “Using correlated subqueries in an UPDATE statement” on page 112
- “Using correlated subqueries in a DELETE statement” on page 113

Correlation

The purpose of a subquery is to supply information needed to evaluate a predicate for a row (WHERE clause) or a group of rows (HAVING clause). This is done through the result table that the subquery produces. Conceptually, the subquery is evaluated whenever a new row or group of rows must be processed. In fact, if the subquery is the same for every row or group, it is evaluated only once. Subqueries like this are said to be **uncorrelated**.

Some subqueries return different values from row to row or group to group. The mechanism that allows this is called **correlation**, and the subqueries are said to be **correlated**. More information specific to correlated subqueries can be found in “Correlated subqueries” on page 109.

Subqueries and search conditions

A subquery can be part of a search condition. The search condition is in the form *operand operator operand*. Either operand can be a subquery. In the following example, the first **operand** is EMPNO and **operator** is IN. The search condition can be part of a WHERE or HAVING clause. The clause can include more than one search condition that contains a subquery. A search condition containing a subquery, like any other search condition, can be enclosed in parentheses, can be preceded by the keyword NOT, and can be linked to other search conditions through the keywords AND and OR. For example, the WHERE clause of a query could look something like this:

```
WHERE (subquery1) = X AND (Y > SOME (subquery2) OR Z = 100)
```


Subqueries can also appear in the search conditions of other subqueries. Such subqueries are said to be **nested** at some level of nesting. For example, a subquery within a subquery within an outer-level SELECT is nested at a nesting level of two. SQL allows nesting down to a nesting level of 32.

How subqueries are used

There are several ways to include a subquery in either a WHERE or HAVING clause:

- “Basic comparisons”
- “Quantified comparisons (ALL, ANY, and SOME)”
- “IN keyword” on page 108
- “EXISTS Keyword” on page 108

Basic comparisons

You can use a subquery before or after any of the comparison operators. The subquery can return at most one value. The value can be the result of a column function or an arithmetic expression. SQL then compares the value that results from the subquery with the value on the other side of the comparison operator. For example, suppose you want to find the employee numbers, names, and salaries for employees whose education level is higher than the average education level throughout the company.

```
SELECT EMPNO, LASTNAME, SALARY
FROM CORPDATA.EMPLOYEE
WHERE EDLEVEL >
      (SELECT AVG(EDLEVEL)
       FROM CORPDATA.EMPLOYEE)
```

SQL first evaluates the subquery and then substitutes the result in the WHERE clause of the SELECT statement. In this example, the result is the company-wide average educational level. Besides returning a single value, a subquery could return no value at all. If it does, the result of the compare is unknown.

Quantified comparisons (ALL, ANY, and SOME)

You can use a subquery after a comparison operator followed by the keyword ALL, ANY, or SOME. When used in this way, the subquery can return zero, one, or many values, including null values. You can use ALL, ANY, and SOME in the following ways:

- Use ALL to indicate that the value you supplied must compare in the indicated way to **ALL** the values the subquery returns. For example, suppose you use the greater-than comparison operator with ALL:

```
... WHERE expression > ALL (subquery)
```

To satisfy this WHERE clause, the value in the expression must be greater than all the values (that is, greater than the highest value) returned by the subquery. If the subquery returns an empty set (that is, no values were selected), the condition is satisfied.

- Use ANY or SOME to indicate that the value you supplied must compare in the indicated way to *at least one* of the values the subquery returns. For example, suppose you use the greater-than comparison operator with ANY:

```
... WHERE expression > ANY (subquery)
```

To satisfy this WHERE clause, the value in the expression must be greater than at least one of the values (that is, greater than the lowest value) returned by the subquery. If what the subquery returns is the empty set, the condition is not satisfied.

Note: The results when a subquery returns one or more null values may surprise you, unless you are familiar with formal logic. For more details, see the discussion of quantified predicates in the SQL Reference.

IN keyword

You can use IN to say that the value in the expression must be among the values returned by the subquery. Using IN is equivalent to using =ANY or =SOME. Using ANY and SOME were previously described. You could also use the IN keyword with the NOT keyword in order to select rows when the value is not among the values returned by the subquery. For example, you could use:

```
... WHERE WORKDEPT NOT IN (SELECT ...)
```

EXISTS Keyword

In the subqueries presented so far, SQL evaluates the subquery and uses the result as part of the WHERE clause of the outer-level SELECT. In contrast, when you use the keyword EXISTS, SQL simply checks whether the subquery returns one or more rows. If it does, the condition is satisfied. If it returns no rows, the condition is not satisfied. For example:

```
SELECT EMPNO, LASTNAME
FROM CORPDATA.EMPLOYEE
WHERE EXISTS
  (SELECT *
   FROM CORPDATA.PROJECT
   WHERE PRSTDATE > '1982-01-01');
```

In the example, the search condition is true if any project represented in the CORPDATA.PROJECT table has an estimated start date that is later than January 1, 1982. Please note that this example does not show the full power of EXISTS, because the result is always the same for every row examined for the outer-level SELECT. As a consequence, either every row appears in the results, or none appear. In a more powerful example, the subquery itself would be correlated, and would change from row to row. See “Correlated subqueries” on page 109 for more information about correlated subqueries.

As shown in the example, you do not need to specify column names in the select-list of the subquery of an EXISTS clause. Instead, you should code SELECT *.

You could also use the EXISTS keyword with the NOT keyword in order to select rows when the data or condition you specify does not exist. You could use the following:

```
... WHERE NOT EXISTS (SELECT ...)
```

Notes on using subqueries

1. When nesting SELECT statements, you can use as many as you need to satisfy your requirements (1 to 31 subqueries), although performance is slower for each additional subquery.
2. When the outer statement is a SELECT statement (at any level of nesting):
 - The subquery can be based on the same table or view as the outer statement, or on a different table or view.
 - You can use a subquery in the WHERE clause of the outer-level SELECT, even when the outer-level SELECT is part of a DECLARE CURSOR, CREATE TABLE, CREATE VIEW, or INSERT statement.
 - You can use a subquery in the HAVING clause of a SELECT statement. When you do, SQL evaluates the subquery and uses it to qualify each group.

3. When the statement is an UPDATE or DELETE statement, you can use subqueries in the WHERE clause of the UPDATE or DELETE statement. You can also use a subquery in the SET clause of an UPDATE statement.
4. When a subquery is used in the SET clause of an UPDATE statement, the result table of a subselect must contain the same number of values as the corresponding list of columns to be updated. In all other cases, the result table for a subquery must consist of a single column, unless the subquery is being used with the EXISTS keyword. For predicates using the keywords ALL, ANY, SOME, or EXISTS, the number of rows returned from the subquery can vary from zero to many. For all other subqueries, the number of rows returned must be zero or one.
5. A subquery cannot include the ORDER BY, UNION, UNION ALL, FOR READ ONLY, FETCH FIRST *n* ROWS, UPDATE, or OPTIMIZE clauses.

Correlated subqueries

In the subqueries previously discussed, SQL evaluates the subquery once, substitutes the result of the subquery in the search condition, and evaluates the outer-level SELECT based on the value of the search condition. You can also write a subquery that SQL may have to re-evaluate as it examines each new row (WHERE clause) or group of rows (HAVING clause) in the outer-level SELECT. This is called a **correlated subquery**.

Correlated names and references

A correlated reference can appear in a search condition in a subquery. The reference is always of the form X.C, where X is a correlation name and C is the name of a column in the table that X represents.

You can define a correlation name for any table appearing in a FROM clause. A correlation name provides a unique name for a table in a query. The same table name can be used many times within a query and its nested subselects. Specifying different correlation names for each table reference makes it possible to uniquely designate which table a column refers to.

The correlation name is defined in the FROM clause of a query. This query could be the outer-level SELECT, or any of the subqueries that contain the one with the reference. Suppose, for example, that a query contains subqueries A, B, and C, and that A contains B and B contains C. Then a correlation name used in C could be defined in B, A, or the outer-level SELECT. To define a correlation name, simply include the correlation name after the table name. Leave one or more blanks between a table name and its correlation name, and place a comma after the correlation name if it is followed by another table name. The following FROM clause defines the correlation names TA and TB for the tables TABLEA and TABLEB, and no correlation name for the table TABLEC.

```
FROM TABLEA TA, TABLEC, TABLEB TB
```

Any number of correlated references can appear in a subquery. For example, one correlated name in a search condition could be defined in the outer-level SELECT, while another could be defined in a containing subquery.

Before the subquery is executed, a value from the referenced column is always substituted for the correlated reference.

Example: Correlated subquery in a WHERE Clause

Suppose that you want a list of all the employees whose education levels are higher than the average education levels in their respective departments. To get this information, SQL must search the CORPDATA.EMPLOYEE table. For each employee in the table, SQL needs to compare the employee's education level to the average education level for the employee's department. In the subquery, you tell SQL to calculate the average education level for the department number in the current row. For example:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM CORPDATA.EMPLOYEE X
WHERE EDLEVEL >
      (SELECT AVG(EDLEVEL)
       FROM CORPDATA.EMPLOYEE
       WHERE WORKDEPT = X.WORKDEPT)
```

A correlated subquery looks like an uncorrelated one, except for the presence of one or more correlated references. In the example, the single correlated reference is the occurrence of X.WORKDEPT in the subselect's FROM clause. Here, the qualifier X is the correlation name defined in the FROM clause of the outer SELECT statement. In that clause, X is introduced as the correlation name of the table CORPDATA.EMPLOYEE.

Now, consider what happens when the subquery is executed for a given row of CORPDATA.EMPLOYEE. Before it is executed, the occurrence of X.WORKDEPT is replaced with the value of the WORKDEPT column for that row. Suppose, for example, that the row is for CHRISTINE I HAAS. Her work department is A00, which is the value of WORKDEPT for this row. The subquery executed for this row is:

```
(SELECT AVG(EDLEVEL)
 FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'A00')
```

Thus, for the row considered, the subquery produces the average education level of Christine's department. This is then compared in the outer statement to Christine's own education level. For some other row for which WORKDEPT has a different value, that value appears in the subquery in place of A00. For example, for the row for MICHAEL L THOMPSON, this value would be B01, and the subquery for his row would deliver the average education level for department B01.

The result table produced by the query would have the following values:

Table 9. Result set for previous query

EMPNO	LASTNAME	WORKDEPT	EDLEVEL
000010	HAAS	A00	18
000030	KWAN	C01	20
000070	PULASKI	D21	16
000090	HENDERSON	E11	16
000110	LUCCHESI	A00	19
000160	PIANKA	D11	17
000180	SCOUTTEN	D11	17
000210	JONES	D11	17
000220	LUTZ	D11	18

Table 9. Result set for previous query (continued)

EMPNO	LASTNAME	WORKDEPT	EDLEVEL
000240	MARINO	D21	17
000260	JOHNSON	D21	16
000280	SCHNEIDER	E11	17
000320	MEHTA	E21	16
000340	GOUNOT	E21	16
200010	HEMMINGER	A00	18
200220	JOHN	D11	18
200240	MONTEVERDE	D21	17
200280	SCHWARTZ	E11	17
200340	ALONZO	E21	16

Example: Correlated subquery in a HAVING Clause

Suppose that you want a list of all the departments whose average salary is higher than the average salary of their area (all departments whose WORKDEPT begins with the same letter belong to the same area). To get this information, SQL must search the CORPDATA.EMPLOYEE table. For each department in the table, SQL compares the department's average salary to the average salary of the area. In the subquery, SQL calculates the average salary for the area of the department in the current group. For example:

```
SELECT WORKDEPT, DECIMAL(AVG(SALARY),8,2)
FROM CORPDATA.EMPLOYEE X
GROUP BY WORKDEPT
HAVING AVG(SALARY) >
  (SELECT AVG(SALARY)
   FROM CORPDATA.EMPLOYEE
   WHERE SUBSTR(X.WORKDEPT,1,1) = SUBSTR(WORKDEPT,1,1))
```

Consider what happens when the subquery is executed for a given department of CORPDATA.EMPLOYEE. Before it is executed, the occurrence of X.WORKDEPT is replaced with the value of the WORKDEPT column for that group. Suppose, for example, that the first group selected has A00 for the value of WORKDEPT. The subquery executed for this group is:

```
(SELECT AVG(SALARY)
 FROM CORPDATA.EMPLOYEE
 WHERE SUBSTR('A00',1,1) = SUBSTR(WORKDEPT,1,1))
```

Thus, for the group considered, the subquery produces the average salary for the area. This value is then compared in the outer statement to the average salary for department 'A00'. For some other group for which WORKDEPT is 'B01', the subquery would result in the average salary for the area where department B01 belongs.

The result table produced by the query would have the following values:

WORKDEPT	AVG SALARY
D21	25668.57
E01	40175.00
E21	24086.66

Example: Correlated subquery in select-list

Suppose that you want a list of all of the departments, including the department name, number, and manager's name. Department names and numbers are found in the CORPDATA.DEPARTMENT table. However, DEPARTMENT only has the manager's number, not the manager's name. To find the name of the manager for each department, we have to find the employee number from the EMPLOYEE table that matches the manager number in the DEPARTMENT table and return the name for the row that matches. Only departments that currently have a manager assigned are to be returned. Execute the following:

```
SELECT DEPTNO, DEPTNAME,  
       (SELECT FIRSTNAME CONCAT ' ' CONCAT  
        MIDINIT CONCAT ' ' CONCAT LASTNAME  
        FROM EMPLOYEE X  
        WHERE X.EMPNO = Y.MGRNO) AS MANAGER_NAME  
FROM DEPARTMENT Y  
WHERE MGRNO IS NOT NULL
```

For each row returned for DEPTNO and DEPTNAME, the system finds where EMPNO = MGRNO and returns the manager's name. The result table produced by the query would have the following values:

Table 10.

DEPTNO	DEPTNAME	MANAGER_NAME
A00	SPIFFY COMPUTER SERVICE DIV.	CHRISTINE I HAAS
B01	PLANNING	MICHAEL L THOMPSON
C01	INFORMATION CENTER	SALLY A KWAN
D11	MANUFACTURING SYSTEMS	IRVING F STERN
D21	ADMINISTRATION SYSTEMS	EVA D PULASKI
E01	SUPPORT SERVICES	JOHN B GEYER
E11	OPERATIONS	EILEEN W HENDERSON
E21	SOFTWARE SUPPORT	THEODORE Q SPENSER

Using correlated subqueries in an UPDATE statement

When you use a correlated subquery in an UPDATE statement, the correlation name refers to the rows you are interested in updating. For example, when all activities of a project must be completed before September 1983, your department considers that project to be a priority project. You could use the SQL statement below to evaluate the projects in the CORPDATA.PROJECT table, and write a 1 (a flag to indicate PRIORITY) in the PRIORITY column (a column you added to CORPDATA.PROJECT for this purpose) for each priority project.

```
UPDATE CORPDATA.PROJECT X  
SET PRIORITY = 1  
WHERE '1983-09-01' >  
      (SELECT MAX(EMENDATE)  
       FROM CORPDATA.EMPPROJACT  
       WHERE PROJNO = X.PROJNO)
```

As SQL examines each row in the CORPDATA.EMPPROJACT table, it determines the maximum activity end date (EMENDATE) for all activities of the project (from the CORPDATA.PROJECT table). If the end date of each activity associated with

the project is prior to September 1983, the current row in the CORPDATA.PROJECT table qualifies and is updated.

Update the master order table with any changes to the quantity ordered. If the quantity in the orders table is not set (the NULL value), keep the value that is in the master order table.

```
UPDATE MASTER_ORDERS X
  SET QTY=(SELECT COALESCE (Y.QTY, X.QTY)
           FROM ORDERS Y
           WHERE X.ORDER_NUM = Y.ORDER_NUM)
 WHERE X.ORDER_NUM IN (SELECT ORDER_NUM
                       FROM ORDERS)
```

In this example, each row of the MASTER_ORDERS table is checked to see if it has a corresponding row in the ORDERS table. If it does have a matching row in the ORDERS table, the COALESCE function is used to return a value for the QTY column. If QTY in the ORDERS table has a non-null value, that value is used to update the QTY column in the MASTER_ORDERS table. If the QTY value in the ORDERS table is NULL, the MASTER_ORDERS QTY column is updated with its own value.

Using correlated subqueries in a DELETE statement

When you use a correlated subquery in a DELETE statement, the correlation name represents the row you delete. SQL evaluates the correlated subquery once for each row in the table named in the DELETE statement to decide whether or not to delete the row.

Suppose a row in the CORPDATA.PROJECT table was deleted. Rows related to the deleted project in the CORPDATA.EMPPROJECT table must also be deleted. To do this, you can use:

```
DELETE FROM CORPDATA.EMPPROJECT X
  WHERE NOT EXISTS
    (SELECT *
     FROM CORPDATA.PROJECT
     WHERE PROJNO = X.PROJNO)
```

SQL determines, for each row in the CORPDATA.EMP_ACT table, whether a row with the same project number exists in the CORPDATA.PROJECT table. If not, the CORPDATA.EMP_ACT row is deleted.

Chapter 8. Sort sequences in SQL

A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. For a complete discussion about sort sequences, see the Sort Sequence section of the SQL Reference book.

The sort sequence is used for all character and UCS-2 graphic comparisons performed in SQL statements. There are sort sequence tables for both single byte and double byte character data. Each single byte sort sequence table has an associated double byte sort sequence table, and vice versa. Conversion between the two tables is performed when necessary to implement a query. In addition, the CREATE INDEX statement has the sort sequence (in effect at the time the statement was run) applied to the character columns referred to in the index.

For more details, see the following topics:

- “Sort sequence used with ORDER BY and row selection”
- “Sort sequence and ORDER BY” on page 116
- “Row selection” on page 117
- “Sort sequence and views” on page 118
- “Sort Sequence and the CREATE INDEX Statement” on page 119
- “Sort sequence and constraints” on page 119

Sort sequence used with ORDER BY and row selection

To see how to use a sort sequence, run the examples in this section against the STAFF table shown in the following table. Notice that the values in the JOB column are in mixed case. You can see the values 'Mgr', 'MGR', and 'mgr'.

Table 11. The STAFF Table

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
20	Pernal	20	Sales	8	18171.25	612.45
30	Merenghi	38	MGR	5	17506.75	0
40	OBrien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	0
60	Quigley	38	SALES	0	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	0	13504.60	128.20
90	Koonitz	42	sales	6	18001.75	1386.70
100	Plotz	42	mgr	6	18352.80	0

In the following examples, the results are shown for each statement using:

- *HEX sort sequence
- Shared-weight sort sequence using the language identifier ENU
- Unique-weight sort sequence using the language identifier ENU

Note: ENU is chosen as a language identifier by specifying either SRTSEQ(*LANGIDUNQ), or SRTSEQ(*LANGIDSHR) and LANGID(ENU), on the CRTSQLxxx, STRSQL, or RUNSQLSTM commands, or by using the SET OPTION statement.

Sort sequence and ORDER BY

The following SQL statement causes the result table to be sorted using the values in the JOB column:

```
SELECT * FROM STAFF ORDER BY JOB
```

Table 12 shows the result table using a *HEX sort sequence. The rows are sorted based on the EBCDIC value in the JOB column. In this case, all lowercase letters sort before the uppercase letters.

*Table 12. "SELECT * FROM STAFF ORDER BY JOB" Using the *HEX Sort Sequence.*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
100	Plotz	42	mgr	6	18352.80	0
90	Koonitz	42	sales	6	18001.75	1386.70
80	James	20	Clerk	0	13504.60	128.20
10	Sanders	20	Mgr	7	18357.50	0
50	Hanes	15	Mgr	10	20659.80	0
30	Merenghi	38	MGR	5	17506.75	0
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55
70	Rothman	15	Sales	7	16502.83	1152.00
60	Quigley	38	SALES	0	16808.30	650.25

Table 13 shows how sorting is done for a unique-weight sort sequence. After the sort sequence is applied to the values in the JOB column, the rows are sorted. Notice that after the sort, lowercase letters are before the same uppercase letters, and the values 'mgr', 'Mgr', and 'MGR' are adjacent to each other.

*Table 13. "SELECT * FROM STAFF ORDER BY JOB" Using the Unique-Weight Sort Sequence for the ENU Language Identifier.*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
80	James	20	Clerk	0	13504.60	128.20
100	Plotz	42	mgr	6	18352.80	0
10	Sanders	20	Mgr	7	18357.50	0
50	Hanes	15	Mgr	10	20659.80	0
30	Merenghi	38	MGR	5	17506.75	0
90	Koonitz	42	sales	6	18001.75	1386.70
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55
70	Rothman	15	Sales	7	16502.83	1152.00
60	Quigley	38	SALES	0	16808.30	650.25

Table 14 shows how sorting is done for a shared-weight sort sequence. After the sort sequence is applied to the values in the JOB column, the rows are sorted. For the sort comparison, each lowercase letter is treated the same as the corresponding uppercase letter. In Table 14, notice that all the values 'MGR', 'mgr' and 'Mgr' are mixed together.

*Table 14. "SELECT * FROM STAFF ORDER BY JOB" Using the Shared-Weight Sort Sequence for the ENU Language Identifier.*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
80	James	20	Clerk	0	13504.60	128.20
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0
100	Plotz	42	mgr	6	18352.80	0
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55
60	Quigley	38	SALES	0	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
90	Koonitz	42	sales	6	18001.75	1386.70

Row selection

The following SQL statement selects rows with the value 'MGR' in the JOB column:

```
SELECT * FROM STAFF WHERE JOB='MGR'
```

Table 15 shows how row selection is done with a *HEX sort sequence. In Table 15, the rows that match the row selection criteria for the column 'JOB' are selected exactly as specified in the select statement. Only the uppercase 'MGR' is selected.

*Table 15. "SELECT * FROM STAFF WHERE JOB='MGR' Using the *HEX Sort Sequence."*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

Table 16 shows how row selection is done with a unique-weight sort sequence. In Table 16, the lowercase and uppercase letters are treated as unique. The lowercase 'mgr' is not treated the same as uppercase 'MGR'. Therefore, the lower case 'mgr' is not selected.

*Table 16. "SELECT * FROM STAFF WHERE JOB = 'MGR' " Using Unique-Weight Sort Sequence for the ENU Language Identifier.*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

Table 17 on page 118 shows how row selection is done with a shared-weight sort sequence. In Table 17 on page 118, the rows that match the row selection criteria for the column 'JOB' are selected by treating uppercase letters the same as lowercase letters. Notice that in Table 17 on page 118 all the values 'mgr', 'Mgr' and 'MGR' are selected.

Table 17. "SELECT * FROM STAFF WHERE JOB = 'MGR' " Using the Shared-Weight Sort Sequence for the ENU Language Identifier.

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0
100	Plotz	42	mgr	6	18352.80	0

Sort sequence and views

Views are created with the sort sequence that was in effect when the CREATE VIEW statement was run. When the view is referred to in a FROM clause, that sort sequence is used for any character comparisons in the subselect of the CREATE VIEW. At that time, an intermediate result table is produced from the view subselect. The sort sequence in effect when the query is being run is then applied to all the character and UCS-2 graphic comparisons (including those comparisons involving implicit conversions to character or UCS-2 graphic) specified in the query.

The following SQL statements and tables show how views and sort sequences work. View V1, used in the following examples, was created with a shared-weight sort sequence of SRTSEQ(*LANGIDSHR) and LANGID(ENU). The CREATE VIEW statement would be as follows:

```
CREATE VIEW V1 AS SELECT *
  FROM STAFF
  WHERE JOB = 'MGR' AND ID < 100
```

Table 18 shows the result table from the view.

Table 18. "SELECT * FROM V1"

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0

Any queries run against view V1 are run against the result table shown in Table 18. The query shown below is run with a sort sequence of SRTSEQ(*LANGIDUNQ) and LANGID(ENU).

Table 19. "SELECT * FROM V1 WHERE JOB = 'MGR'" Using the Unique-Weight Sort Sequence for Language Identifier ENU

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

Sort Sequence and the CREATE INDEX Statement

Indexes are created using the sort sequence that was in effect when the CREATE INDEX statement was run. An entry is added to the index every time an insert is made into the table over which the index is defined. Index entries contain the weighted value for character key and UCS-2 graphic key columns. The system gets the weighted value by converting the key value based on the sort sequence of the index.

When selection is made using that sort sequence and that index, the character or UCS-2 graphic keys do not need to be converted prior to comparison. This improves the performance of the query. For more information about creating effective indexes and sort sequence, see *Using indexes to speed access to large tables* in the *Database Performance and Query Optimization* book.

Sort sequence and constraints

Unique constraints are implemented with indexes. If the table on which a unique constraint is added was defined with a sort sequence, the index will be created with that same sort sequence.

If defining a referential constraint, the sort sequence between the parent and dependent table must match. For more information about sort sequence and constraints, see the *Ensuring data integrity with referential constraints* topic in the *Database Programming* book in the iSeries Information Center.

The sort sequence used at the time a check constraint is defined is the same sort sequence the system uses to validate adherence to the constraint at the time of an INSERT or UPDATE.

Chapter 9. Using a Cursor

When SQL runs a select statement, the resulting rows comprise the result table. A cursor provides a way to access a result table. It is used within an SQL program to maintain a position in the result table. SQL uses a cursor to work with the rows in the result table and to make them available to your program. Your program can have several cursors, although each must have a unique name.

Statements related to using a cursor include the following:

- A DECLARE CURSOR statement to define and name the cursor and specify the rows to be retrieved with the embedded select statement.
- OPEN and CLOSE statements to open and close the cursor for use within the program. The cursor must be opened before any rows can be retrieved.
- A FETCH statement to retrieve rows from the cursor's result table or to position the cursor on another row.
- An UPDATE ... WHERE CURRENT OF statement to update the current row of a cursor.
- A DELETE ... WHERE CURRENT OF statement to delete the current row of a cursor.

For a complete discussion of these statements, see the SQL Reference book.

See the following topics for more information about cursors:

- "Types of cursors"
- "Example of using a cursor" on page 122
- "Using the multiple-row FETCH statement" on page 128
- "Unit of work and open cursors" on page 133

Note: See "Code disclaimer information" on page x information for information pertaining to code examples.

Types of cursors

SQL supports serial and scrollable cursors. The type of cursor determines the positioning methods which can be used with the cursor. For more information, see:

- "Serial cursor"
- "Scrollable cursor" on page 122

Serial cursor

A serial cursor is one defined without the SCROLL keyword.

For a serial cursor, each row of the result table can be fetched only once per OPEN of the cursor. When the cursor is opened, it is positioned before the first row in the result table. When a FETCH is issued, the cursor is moved to the next row in the result table. That row is then the current row. If host variables are specified (with the INTO clause on the FETCH statement), SQL moves the current row's contents into your program's host variables.

This sequence is repeated each time a FETCH statement is issued until the end-of-data (SQLCODE = 100) is reached. When you reach the end-of-data, close the cursor. You cannot access any rows in the result table after you reach the end-of-data. To use a serial cursor again, you must first close the cursor and then re-issue the OPEN statement. You can never back up using a serial cursor.

Scrollable cursor

For a scrollable cursor, the rows of the result table can be fetched many times. The cursor is moved through the result table based on the position option specified on the FETCH statement. When the cursor is opened, it is positioned before the first row in the result table. When a FETCH is issued, the cursor is positioned to the row in the result table that is specified by the position option. That row is then the current row. If host variables are specified (with the INTO clause on the FETCH statement), SQL moves the current row's contents into your program's host variables. Host variables cannot be specified for the BEFORE and AFTER position options.

This sequence is repeated each time a FETCH statement is issued. The cursor does not need to be closed when an end-of-data or beginning-of-data condition occurs. The position options enable the program to continue fetching rows from the table.

The following scroll options are used to position the cursor when issuing a FETCH statement. These positions are relative to the current cursor location in the result table.

NEXT	Positions the cursor on the next row. This is the default if no position is specified.
PRIOR	Positions the cursor on the previous row.
FIRST	Positions the cursor on the first row.
LAST	Positions the cursor on the last row.
BEFORE	Positions the cursor before the first row.
AFTER	Positions the cursor after the last row.
CURRENT	Does not change the cursor position.
RELATIVE n	Evaluates a host variable or integer <i>n</i> in relationship to the cursor's current position. For example, if <i>n</i> is -1, the cursor is positioned on the previous row of the result table. If <i>n</i> is +3, the cursor is positioned three rows after the current row.

For a scrollable cursor, the end of the table can be determined by the following:

```
FETCH AFTER FROM C1
```

Once the cursor is positioned at the end of the table, the program can use the PRIOR or RELATIVE scroll options to position and fetch data starting from the end of the table.

Example of using a cursor

Suppose your program examines data about people in department D11. The following examples show the SQL statements you would include in a program to define and use a serial and a scrollable cursor. These cursors can be used to obtain information about the department from the CORPDATA.EMPLOYEE table.

For the serial cursor example, the program processes all of the rows from the table, updating the job for all members of department D11 and deleting the records of employees from the other departments.

Table 20. A Serial Cursor Example

Serial Cursor SQL Statement	Described in Section
EXEC SQL DECLARE THISEMP CURSOR FOR SELECT EMPNO, LASTNAME, WORKDEPT, JOB FROM CORPDATA.EMPLOYEE FOR UPDATE OF JOB END-EXEC.	“Step 1: Define the cursor” on page 124.
EXEC SQL OPEN THISEMP END-EXEC.	“Step 2: Open the cursor” on page 126.
EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.	“Step 3: Specify what to do when end-of-data is reached” on page 126.
EXEC SQL FETCH THISEMP INTO :EMP-NUM, :NAME2, :DEPT, :JOB-CODE END-EXEC.	“Step 4: Retrieve a row using a cursor” on page 126.
... for all employees in department D11, update the JOB value:	“Step 5a: Update the current row” on page 127.
EXEC SQL UPDATE CORPDATA.EMPLOYEE SET JOB = :NEW-CODE WHERE CURRENT OF THISEMP END-EXEC.	
... then print the row.	
... for other employees, delete the row:	“Step 5b: Delete the current row” on page 127.
EXEC SQL DELETE FROM CORPDATA.EMPLOYEE WHERE CURRENT OF THISEMP END-EXEC.	
Branch back to fetch and process the next row.	
CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.	“Step 6: Close the cursor” on page 128.

For the scrollable cursor example, the program uses the RELATIVE position option to obtain a representative sample of salaries from department D11.

Table 21. Scrollable Cursor Example

Scrollable Cursor SQL Statement	Described in Section
<pre>EXEC SQL DECLARE THISEMP DYNAMIC SCROLL CURSOR FOR SELECT EMPNO, LASTNAME, SALARY FROM CORPDATA.EMPLOYEE WHERE WORKDEPT = 'D11' END-EXEC.</pre>	"Step 1: Define the cursor".
<pre>EXEC SQL OPEN THISEMP END-EXEC.</pre>	"Step 2: Open the cursor" on page 126.
<pre>EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.</pre>	"Step 3: Specify what to do when end-of-data is reached" on page 126.
<pre>...initialize program summation salary variable EXEC SQL FETCH RELATIVE 3 FROM THISEMP INTO :EMP-NUM, :NAME2, :JOB-CODE END-EXEC. ...add the current salary to program summation salary ...branch back to fetch and process the next row.</pre>	"Step 4: Retrieve a row using a cursor" on page 126.
<pre>...calculate the average salary</pre>	
<pre>CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.</pre>	"Step 6: Close the cursor" on page 128.

Step 1: Define the cursor

To define a result table to be accessed with a cursor, use the DECLARE CURSOR statement.

The DECLARE CURSOR statement names a cursor and specifies a select-statement. The select-statement defines a set of rows that, conceptually, make up the result table. For a serial cursor, the statement looks like this (the FOR UPDATE OF clause is optional):

```
EXEC SQL
  DECLARE cursor-name CURSOR FOR
  SELECT column-1, column-2 ,...
  FROM table-name , ...
  FOR UPDATE OF column-2 ,...
END-EXEC.
```

For a scrollable cursor, the statement looks like this (the WHERE clause is optional):

```

EXEC SQL
  DECLARE cursor-name DYNAMIC SCROLL CURSOR FOR
  SELECT column-1, column-2 ,...
  FROM table-name ,...
  WHERE column-1 = expression ...
END-EXEC.

```

The select-statements shown here are rather simple. However, you can code several other types of clauses in a select-statement within a DECLARE CURSOR statement for a serial and a scrollable cursor.

If you intend to update any columns in any or all of the rows of the identified table (the table named in the FROM clause), include the FOR UPDATE OF clause. It names each column you intend to update. If you do not specify the names of columns, and you specify either the ORDER BY clause or FOR READ ONLY clause, a negative SQLCODE is returned if an update is attempted. If you do not specify the FOR UPDATE OF clause, the FOR READ ONLY clause, or the ORDER BY clause, and the result table is not read-only, you can update any of the columns of the specified table.

You can update a column of the identified table even though it is not part of the result table. In this case, you do not need to name the column in the SELECT statement. When the cursor retrieves a row (using FETCH) that contains a column value you want to update, you can use UPDATE ... WHERE CURRENT OF to update the row.

For example, assume that each row of the result table includes the *EMPNO*, *LASTNAME*, and *WORKDEPT* columns from the *CORPDATA.EMPLOYEE* table. If you want to update the *JOB* column (one of the columns in each row of the *CORPDATA.EMPLOYEE* table), the DECLARE CURSOR statement should include FOR UPDATE OF JOB ... even though JOB is omitted from the SELECT statement.

The result table and cursor are *read-only* if any of the following are true:

- The first FROM clause identifies more than one table or view.
- The first FROM clause identifies a read-only view.
- The first FROM clause identifies a user-defined table function.
- The first SELECT clause specifies the keyword DISTINCT.
- The outer subselect contains a GROUP BY clause.
- The outer subselect contains a HAVING clause.
- The first SELECT clause contains a column function.
- The select-statement contains a subquery such that the base object of the outer subselect and of the subquery is the same table.
- The select-statement contains a UNION or UNION ALL operator.
- The select-statement contains an ORDER BY clause, and the FOR UPDATE OF clause and DYNAMIC SCROLL are not specified.
- The select-statement includes a FOR READ ONLY clause.
- The SCROLL keyword is specified without DYNAMIC.
- The select-list includes a DataLink column and a FOR UPDATE OF clause is not specified.
- The first subselect requires a temporary result table.
- The select-statement includes a FETCH FIRST *n* ROWS ONLY.

Step 2: Open the cursor

To begin processing the rows of the result table, issue the OPEN statement. When your program issues the OPEN statement, SQL processes the select-statement within the DECLARE CURSOR statement to identify a set of rows, called a result table, using the current value of any host variables specified in the select-statement. A result table can contain zero, one, or many rows, depending on the extent to which the search condition is satisfied. The OPEN statement looks like this:

```
EXEC SQL
  OPEN cursor-name
END-EXEC.
```

Step 3: Specify what to do when end-of-data is reached

To find out when the end of the result table is reached, test the SQLCODE field for a value of 100 or test the SQLSTATE field for a value of '02000' (that is, end-of-data). This condition occurs when the FETCH statement has retrieved the last row in the result table and your program issues a subsequent FETCH. For example:

```
...
IF SQLCODE =100 GO TO DATA-NOT-FOUND.
```

or

```
IF SQLSTATE ='02000' GO TO DATA-NOT-FOUND.
```

An alternative to this technique is to code the WHENEVER statement. Using WHENEVER NOT FOUND can result in a branch to another part of your program, where a CLOSE statement is issued. The WHENEVER statement looks like this:

```
EXEC SQL
  WHENEVER NOT FOUND GO TO symbolic-address
END-EXEC.
```

Your program should anticipate an end-of-data condition whenever a cursor is used to fetch a row, and should be prepared to handle this situation when it occurs.

When you are using a serial cursor and the end-of-data is reached, every subsequent FETCH statement returns the end-of-data condition. You cannot position the cursor on rows that are already processed. The CLOSE statement is the only operation that can be performed on the cursor.

When you are using a scrollable cursor and the end-of-data is reached, the result table can still process more data. You can position the cursor anywhere in the result table using a combination of the position options. You do not need to CLOSE the cursor when the end-of-data is reached.

Step 4: Retrieve a row using a cursor

To move the contents of a selected row into your program's host variables, use the FETCH statement. The SELECT statement within the DECLARE CURSOR statement identifies rows that contain the column values your program wants. However, SQL does not retrieve any data for your application program until the FETCH statement is issued.

When your program issues the `FETCH` statement, SQL uses the current cursor position as a starting point to locate the requested row in the result table. This changes that row to the **current row**. If an `INTO` clause was specified, SQL moves the current row's contents into your program's host variables. This sequence is repeated each time the `FETCH` statement is issued.

SQL maintains the position of the current row (that is, the cursor points to the current row) until the next `FETCH` statement for the cursor is issued. The `UPDATE` statement does not change the position of the current row within the result table, although the `DELETE` statement does.

The serial cursor `FETCH` statement looks like this:

```
EXEC SQL
  FETCH cursor-name
    INTO :host variable-1[, :host variable-2] ...
END-EXEC.
```

The scrollable cursor `FETCH` statement looks like this:

```
EXEC SQL
  FETCH RELATIVE integer
    FROM cursor-name
    INTO :host variable-1[, :host variable-2] ...
END-EXEC.
```

Step 5a: Update the current row

When your program has positioned the cursor on a row, you can update its data by using the `UPDATE` statement with the `WHERE CURRENT OF` clause. The `WHERE CURRENT OF` clause specifies a cursor that points to the row you want to update. The `UPDATE ... WHERE CURRENT OF` statement looks like this:

```
EXEC SQL
  UPDATE table-name
    SET column-1 = value [, column-2 = value] ...
    WHERE CURRENT OF cursor-name
END-EXEC.
```

When used with a cursor, the `UPDATE` statement:

- Updates only one row—the current row
- Identifies a cursor that points to the row to be updated
- Requires that the columns updated be named previously in the `FOR UPDATE OF` clause of the `DECLARE CURSOR` statement, if an `ORDER BY` clause was also specified

After you update a row, the cursor's position remains on that row (that is, the current row of the cursor does not change) until you issue a `FETCH` statement for the next row.

Step 5b: Delete the current row

When your program has retrieved the current row, you can delete the row by using the `DELETE` statement. To do this, you issue a `DELETE` statement designed for use with a cursor; the `WHERE CURRENT OF` clause specifies a cursor that points to the row you want to delete. The `DELETE ... WHERE CURRENT OF` statement looks like this:

```
EXEC SQL
  DELETE FROM table-name
    WHERE CURRENT OF cursor-name
END-EXEC.
```

When used with a cursor, the DELETE statement:

- Deletes only one row—the current row
- Uses the WHERE CURRENT OF clause to identify a cursor that points to the row to be deleted

After you delete a row, you cannot update or delete another row using that cursor until you issue a FETCH statement to position the cursor.

“Removing rows from a table using the DELETE statement” on page 102 shows you how to use the DELETE statement to delete all rows that meet a specific search condition. You can also use the FETCH and DELETE ... WHERE CURRENT OF statements when you want to obtain a copy of the row, examine it, then delete it.

Step 6: Close the cursor

If you processed the rows of a result table for a serial cursor, and you want to use the cursor again, issue a CLOSE statement to close the cursor prior to re-opening it.

```
EXEC SQL
  CLOSE cursor-name
END-EXEC.
```

If you processed the rows of a result table and you do not want to use the cursor again, you can let the system close the cursor. The system automatically closes the cursor when:

- A COMMIT without HOLD statement is issued and the cursor is not declared using the WITH HOLD clause.
- A ROLLBACK without HOLD statement is issued.
- The job ends.
- The activation group ends and CLOSQLCSR(*ENDACTGRP) was specified on the precompile.
- The first SQL program in the call stack ends and neither CLOSQLCSR(*ENDJOB) or CLOSQLCSR(*ENDACTGRP) was specified when the program was precompiled.
- The connection to the application server is ended using the DISCONNECT statement.
- The connection to the application server was released and a successful COMMIT occurred.
- An *RUW CONNECT occurred.

Because an open cursor still holds locks on referred-to-tables or views, you should explicitly close any open cursors as soon as they are no longer needed.

Using the multiple-row FETCH statement

The multiple-row FETCH statement can be used to retrieve multiple rows from a table or view with a single FETCH. The program controls the blocking of rows by the number of rows requested on the FETCH statement (OVRDBF has no effect). The maximum number of rows that can be requested on a single fetch call is 32767. Once the data is retrieved, the cursor is positioned on the last row retrieved.

There are two ways to define the storage where fetched rows are placed: a host structure array or a row storage area with an associated descriptor. Both methods can be coded in all of the languages supported by the SQL precompilers, with the

exception of the host structure array in REXX. Refer to the SQL Programming with Host Languages information for details on the programming languages. Both forms of the multiple-row FETCH statement allow the application to code a separate indicator array. The indicator array should contain one indicator for each host variable that is null capable.

The multiple-row FETCH statement can be used with both serial and scrollable cursors. The operations used to define, open, and close a cursor for a multiple-row FETCH remain the same. Only the FETCH statement changes to specify the number of rows to retrieve and the storage where the rows are placed.

After each multiple-row FETCH, information is returned to the program through the SQLCA. In addition to the SQLCODE and SQLSTATE fields, the SQLERRD provides the following information:

- SQLERRD3 contains the number of rows retrieved on the multiple-row FETCH statement. If SQLERRD3 is less than the number of rows requested, then an error or end-of-data condition occurred.
- SQLERRD4 contains the length of each row retrieved.
- SQLERRD5 contains an indication that the last row in the table was fetched. It can be used to detect the end-of-data condition in the table being fetched when the cursor does not have immediate sensitivity to updates. Cursors which do have immediate sensitivity to updates should continue fetching until an SQLCODE +100 is received to detect an end-of-data condition.

Multiple-row FETCH using a host structure array

To use the multiple-row FETCH with the host structure array, the application must define a host structure array that can be used by SQL. Each language has its own conventions and rules for defining a host structure array. Host structure arrays can be defined by using variable declarations or by using compiler directives to retrieve External File Descriptions (such as the COBOL COPY directive).

The host structure array consists of an array of structures. Each structure corresponds to one row of the result table. The first structure in the array corresponds to the first row, the second structure in the array corresponds to the second row, and so on. SQL determines the attributes of elementary items in the host structure array based on the declaration of the host structure array. To maximize performance, the attributes of the items that make up the host structure array should match the attributes of the columns being retrieved.

Consider the following COBOL example:

```
EXEC SQL INCLUDE SQLCA
END-EXEC.

...

01 TABLE-1.
  02 DEPT OCCURS 10 TIMES.
    05 EMPNO PIC X(6).
    05 LASTNAME.
      49 LASTNAME-LEN PIC S9(4) BINARY.
      49 LASTNAME-TEXT PIC X(15).
    05 WORKDEPT PIC X(3).
    05 JOB PIC X(8).
01 TABLE-2.
  02 IND-ARRAY OCCURS 10 TIMES.
    05 INDS PIC S9(4) BINARY OCCURS 4 TIMES.
```

```

...
EXEC SQL
DECLARE D11 CURSOR FOR
SELECT EMPNO, LASTNAME, WORKDEPT, JOB
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = "D11"
END-EXEC.

...

EXEC SQL
OPEN D11
END-EXEC.
PERFORM FETCH-PARA UNTIL SQLCODE NOT EQUAL TO ZERO.
ALL-DONE.
EXEC SQL CLOSE D11 END-EXEC.

...

FETCH-PARA.
EXEC SQL WHENEVER NOT FOUND GO TO ALL-DONE END-EXEC.
EXEC SQL FETCH D11 FOR 10 ROWS INTO :DEPT :IND-ARRAY
END-EXEC.

...

```

In this example, a cursor was defined for the CORPDATA.EMPLOYEE table to select all rows where the WORKDEPT column equals 'D11'. The result table contains eight rows. The DECLARE CURSOR and OPEN statements do not have any special syntax when they are used with a multiple-row FETCH statement. Another FETCH statement that returns a single row against the same cursor can be coded elsewhere in the program. The multiple-row FETCH statement is used to retrieve all of the rows in the result table. Following the FETCH, the cursor position remains on the last row retrieved.

The host structure array DEPT and the associated indicator array IND-ARRAY are defined in the application. Both arrays have a dimension of ten. The indicator array has an entry for each column in the result table.

The attributes of type and length of the DEPT host structure array elementary items match the columns that are being retrieved.

When the multiple-row FETCH statement has successfully completed, the host structure array contains the data for all eight rows. The indicator array, IND_ARRAY, contains zeros for every column in every row because no NULL values were returned.

The SQLCA that is returned to the application contains the following information:

- SQLCODE contains 0
- SQLSTATE contains '00000'
- SQLERRD3 contains 8, the number of rows fetched
- SQLERRD4 contains 34, the length of each row
- SQLERRD5 contains +100, indicating the last row in the result table is in the block

See Appendix B of the SQL Reference book for a description of the SQLCA.

Multiple-row FETCH using a row storage area

The application must define a row storage area and an associated description area before the application can use a multiple-row FETCH with a row storage area. The row storage area is a host variable defined in the application program. The row storage area contains the results of the multiple-row FETCH. A row storage area can be a character variable with enough bytes to hold all of the rows requested on the multiple-row FETCH.

An SQLDA that contains the SQLTYPE and SQLLEN for each returned column is defined by the associated descriptor used on the row storage area form of the multiple-row FETCH. The information provided in the descriptor determines the data mapping from the database to the row storage area. To maximize performance, the attribute information in the descriptor should match the attributes of the columns retrieved.

See Appendix C of the SQL Reference book for a description of the SQLDA.

Consider the following PL/I example:

```
*.....1.....2.....3.....4.....5.....6.....7...*
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;

...

DCL DEPTPTR PTR;
DCL 1 DEPT(20) BASED(DEPTPTR),
    3 EMPNO CHAR(6),
    3 LASTNAME CHAR(15) VARYING,
    3 WORKDEPT CHAR(3),
    3 JOB CHAR(8);
DCL I BIN(31) FIXED;
DEC J BIN(31) FIXED;
DCL ROWAREA CHAR(2000);

...

ALLOCATE SQLDA SET(SQLDAPTR);
EXEC SQL
  DECLARE D11 CURSOR FOR
  SELECT EMPNO, LASTNAME, WORKDEPT, JOB
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'D11';
```

Figure 1. Example of Multiple-Row FETCH Using a Row Storage Area (Part 1 of 2)

```

...
EXEC SQL
  OPEN D11;
/* SET UP THE DESCRIPTOR FOR THE MULTIPLE-ROW FETCH */
/* 4 COLUMNS ARE BEING FETCHED */
SQLD = 4;
SQLN = 4;
SQLDABC = 366;
SQLTYPE(1) = 452; /* FIXED LENGTH CHARACTER - */
/* NOT NULLABLE */
SQLLEN(1) = 6;
SQLTYPE(2) = 456; /*VARYING LENGTH CHARACTER */
/* NOT NULLABLE */
SQLLEN(2) = 15;
SQLTYPE(3) = 452; /* FIXED LENGTH CHARACTER - */
SQLLEN(3) = 3;
SQLTYPE(4) = 452; /* FIXED LENGTH CHARACTER - */
/* NOT NULLABLE */
SQLLEN(4) = 8;
/*ISSUE THE MULTIPLE-ROW FETCH STATEMENT TO RETRIEVE*/
/*THE DATA INTO THE DEPT ROW STORAGE AREA */
/*USE A HOST VARIABLE TO CONTAIN THE COUNT OF */
/*ROWS TO BE RETURNED ON THE MULTIPLE-ROW FETCH */
J = 20; /*REQUESTS 20 ROWS ON THE FETCH */
...
EXEC SQL
  WHENEVER NOT FOUND
  GOTO FINISHED;
EXEC SQL
  WHENEVER SQLERROR
  GOTO FINISHED;
EXEC SQL
  FETCH D11 FOR :J ROWS
  USING DESCRIPTOR :SQLDA INTO :ROWAREA;
/* ADDRESS THE ROWS RETURNED */
DEPTPTR = ADDR(ROWAREA);
/*PROCESS EACH ROW RETURNED IN THE ROW STORAGE */
/*AREA BASED ON THE COUNT OF RECORDS RETURNED */
/*IN SQLERRD3. */
DO I = 1 TO SQLERRD(3);
  IF EMPNO(I) = '000170' THEN
    DO;
  :
  END;
END;
IF SQLERRD(5) = 100 THEN
  DO;
  /* PROCESS END OF FILE */
  END;
FINISHED:

```

Figure 1. Example of Multiple-Row FETCH Using a Row Storage Area (Part 2 of 2)

In this example, a cursor has been defined for the CORPDATA.EMPLOYEE table to select all rows where the WORKDEPT column equal 'D11'. The sample EMPLOYEE table in Appendix A, “DB2 UDB for iSeries Sample Tables” shows the result table contains multiple rows. The DECLARE CURSOR and OPEN statements do not have special syntax when they are used with a multiple-row FETCH statement. Another FETCH statement that returns a single row against the same cursor can be coded elsewhere in the program. The multiple-row FETCH statement is used to retrieve all rows in the result table. Following the FETCH, the cursor position remains on the final row in the block.

The row area, ROWAREA, is defined as a character array. The data from the result table is placed in the host variable. In this example, a pointer variable is assigned to the address of ROWAREA. Each item in the rows that are returned is examined and used with the based structure DEPT.

The attributes (type and length) of the items in the descriptor match the columns that are retrieved. In this case, no indicator area is provided.

After the FETCH statement is completed, the ROWAREA contains all of the rows that equal 'D11', in this case 11 rows. The SQLCA that is returned to the application contains the following:

- SQLCODE contains 0
- SQLSTATE contains '00000'
- SQLERRD3 contains 11, the number of rows returned
- SQLERRD4 contains 34, for the length of the row fetched
- SQLERRD5 contains +100, indicating the last row in the result table was fetched

In this example, the application has taken advantage of the fact that SQLERRD5 contains an indication of the end of the file being reached. As a result, the application does not need to call SQL again to attempt to retrieve more rows. If the cursor has immediate sensitivity to inserts, you should call SQL in case any records were added. Cursors have immediate sensitivity when the commitment control level is something other than *RR.

Unit of work and open cursors

When your program completes a unit of work, it should commit or rollback the changes you made. Unless you specified HOLD on the COMMIT or ROLLBACK statement, all open cursors are automatically closed by SQL. Cursors that are declared with the WITH HOLD clause are not automatically closed on COMMIT. They are automatically closed on a ROLLBACK (the WITH HOLD clause specified on the DECLARE CURSOR statement is ignored).

If you want to continue processing from the current cursor position after a COMMIT or ROLLBACK, you must specify COMMIT HOLD or ROLLBACK HOLD. When HOLD is specified, any open cursors are left open and keep their cursor position so processing can resume. On a COMMIT statement, the cursor position is maintained. On a ROLLBACK statement, the cursor position is restored to just after the last row retrieved from the previous unit of work. All record locks are still released.

After issuing a COMMIT or ROLLBACK statement without HOLD, all locks are released and all cursors are closed. You can open the cursor again, but you will begin processing at the first row of the result table.

Note: Specification of the ALWBLK(*ALLREAD) parameter of the CRTSQLxxx commands can change the restoration of the cursor position for read-only cursors. See Chapter 14, "Dynamic SQL Applications" for information on the use of the ALWBLK parameter and other performance related options on the CRTSQLxxx commands.

For more information about commitment control and unit of work, see the Commitment control topic.

Chapter 10. Data Integrity

Data integrity is the principle of ensuring data values between tables of a schema are kept in a state that makes sense to the business. For example, if a bank has a list of customers in table A and a list of customer accounts in table B, it would not make sense to allow a new account to be added to table B unless an associated customer exists in table A.

This chapter describes the different ways the system automatically enforces these kinds of relationships. Referential integrity, check constraints, and triggers are all ways of accomplishing data integrity. Additionally, the `WITH CHECK OPTION` clause on a `CREATE VIEW` constrains the inserting or updating of data through a view. For more information, see the following topics:

- “Adding and using check constraints”
- “Referential integrity” on page 136
- “WITH CHECK OPTION on a View” on page 144
- “DB2 UDB for iSeries Trigger support” on page 147

For comprehensive information about data integrity, see the Database Programming book.

Adding and using check constraints

A *check constraint* assures the validity of data during inserts and updates by limiting the allowable values in a column or group of columns. Use the SQL `CREATE TABLE` and `ALTER TABLE` statements to add or drop check constraints.

In this example, the following statement creates a table with three columns and a check constraint over `COL2` that limits the values allowed in that column to positive integers:

```
CREATE TABLE T1 (COL1 INT, COL2 INT CHECK (COL2>0), COL3 INT)
```

Given this table, the following statement:

```
INSERT INTO T1 VALUES (-1, -1, -1)
```

would fail because the value to be inserted into `COL2` does not meet the check constraint; that is, `-1` is not greater than `0`.

The following statement would be successful:

```
INSERT INTO T1 VALUES (1, 1, 1)
```

Once that row is inserted, the following statement would fail:

```
ALTER TABLE T1 ADD CONSTRAINT C1 CHECK (COL1=1 AND COL1<COL2)
```

This `ALTER TABLE` statement attempts to add a second check constraint that limits the value allowed in `COL1` to `1` and also effectively rules that values in `COL2` be greater than `1`. This constraint would not be allowed because the second part of the constraint is not met by the existing data (the value of `'1'` in `COL2` is not less than the value of `'1'` in `COL1`).

Referential integrity

Referential integrity is the condition of a set of tables in a database in which all references from one table to another are valid.

Consider the following example: (These sample tables are given in Appendix A, “DB2 UDB for iSeries Sample Tables”):

- CORPDATA.EMPLOYEE serves as a master list of employees.
- CORPDATA.DEPARTMENT acts as a master list of all valid department numbers.
- CORPDATA.EMP_ACT provides a master list of activities performed for projects.

Other tables refer to the same entities described in these tables. When a table contains data for which there is a master list, that data should actually appear in the master list, or the reference is not valid. The table that contains the master list is the *parent table*, and the table that refers to it is a *dependent table*. When the references from the dependent table to the parent table are valid, the condition of the set of tables is called *referential integrity*.

Stated another way, referential integrity is the state of a database in which all values of all foreign keys are valid. Each value of the foreign key must also exist in the parent key or be null. This definition of referential integrity requires an understanding of the following terms:

- A *unique key* is a column or set of columns in a table which uniquely identify a row. Although a table can have several unique keys, no two rows in a table can have the same unique key value.
- A *primary key* is a unique key that does not allow nulls. A table cannot have more than one primary key.
- A *parent key* is either a unique key or a primary key which is referenced in a referential constraint.
- A *foreign key* is a column or set of columns whose values must match those of a parent key. If any column value used to build the foreign key is null, then the rule does not apply.
- A *parent table* is a table that contains the parent key.
- A *dependent table* is the table that contains the foreign key.
- A *descendent table* is a table that is a dependent table or a descendent of a dependent table.

Enforcement of referential integrity prevents the violation of the rule which states that every non-null foreign key must have a matching parent key.

For more information about referential integrity, see the following topics:

- “Adding or dropping referential constraints” on page 137
- “Removing referential constraints” on page 138
- “Inserting into tables with referential constraints” on page 138
- “Updating tables with referential constraints” on page 140
- “Deleting from tables with referential constraints” on page 141
- “Check pending” on page 144

SQL supports the referential integrity concept with the CREATE TABLE and ALTER TABLE statements. For detailed descriptions of these commands, see the

SQL Reference book. You can also add constraints using when creating a table or add them to an existing table using iSeries Navigator.

Adding or dropping referential constraints

Constraints are rules that ensure that references from one table, a dependent table, to data in another table, the parent table, are valid. You use referential constraints to ensure Referential integrity.

Use the SQL CREATE TABLE and ALTER TABLE statements to add or change referential constraints.

With a referential constraint, non-null values of the foreign key are valid only if they also appear as values of a parent key. When you define a referential constraint, you specify:

- A primary or unique key
- A foreign key
- Delete and update rules that specify the action taken with respect to dependent rows when the parent row is deleted or updated.

Optionally, you can specify a name for the constraint. If a name is not specified, one is automatically generated.

Once a referential constraint is defined, the system enforces the constraint on every INSERT, DELETE, and UPDATE operation performed through SQL or any other interface including iSeries Navigator, CL commands, utilities, or high-level language statements.

Example: Adding referential constraints

The rule that every department number shown in the sample employee table must appear in the department table is a referential constraint. This constraint ensures that every employee belongs to an existing department. The following SQL statements create the CORPDATA.DEPARTMENT and CORPDATA.EMPLOYEE tables with those constraint relationships defined.

```
CREATE TABLE CORPDATA.DEPARTMENT
(DEPTNO    CHAR(3)    NOT NULL PRIMARY KEY,
DEPTNAME  VARCHAR(29) NOT NULL,
MGRNO     CHAR(6),
ADMRDEPT  CHAR(3)    NOT NULL
CONSTRAINT REPORTS_TO_EXISTS
REFERENCES CORPDATA.DEPARTMENT (DEPTNO)
ON DELETE CASCADE)
```

```
CREATE TABLE CORPDATA.EMPLOYEE
(EMPNO     CHAR(6)    NOT NULL PRIMARY KEY,
FIRSTNAME VARCHAR(12) NOT NULL,
MIDINIT   CHAR(1)    NOT NULL,
LASTNAME  VARCHAR(15) NOT NULL,
WORKDEPT  CHAR(3)    CONSTRAINT WORKDEPT_EXISTS
REFERENCES CORPDATA.DEPARTMENT (DEPTNO)
ON DELETE SET NULL ON UPDATE RESTRICT,

PHONENO   CHAR(4),
HIREDATE  DATE,
JOB       CHAR(8),
EDLEVEL   SMALLINT   NOT NULL,
SEX       CHAR(1),
BIRTHDATE DATE,
SALARY    DECIMAL(9,2),
```

```
BONUS    DECIMAL(9,2),
COMM     DECIMAL(9,2),
CONSTRAINT UNIQUE_LNAME_IN_DEPT UNIQUE (WORKDEPT, LASTNAME))
```

In this case, the DEPARTMENT table has a column of unique department numbers (DEPTNO) which functions as a primary key, and is a parent table in two constraint relationships:

REPORTS_TO_EXISTS

is a self-referencing constraint in which the DEPARTMENT table is both the parent and the dependent in the same relationship. Every non-null value of ADMRDEPT must match a value of DEPTNO. A department must report to an existing department in the database. The DELETE CASCADE rule indicates that if a row with a DEPTNO value *n* is deleted, every row in the table for which the ADMRDEPT is *n* is also deleted.

WORKDEPT_EXISTS

establishes the EMPLOYEE table as a dependent table, and the column of employee department assignments (WORKDEPT) as a foreign key. Thus, every value of WORKDEPT must match a value of DEPTNO. The DELETE SET NULL rule says that if a row is deleted from DEPARTMENT in which the value of DEPTNO is *n*, then the value of WORKDEPT in EMPLOYEE is set to null in every row in which the value was *n*. The UPDATE RESTRICT rule says that a value of DEPTNO in DEPARTMENT cannot be updated if there are values of WORKDEPT in EMPLOYEE that match the current DEPTNO value.

Constraint UNIQUE_LNAME_IN_DEPT in the EMPLOYEE table causes last names to be unique within a department. While this constraint is unlikely, it illustrates how a constraint made up of several columns can be defined at the table level.

Removing referential constraints

The ALTER TABLE statement can be used to add or drop one constraint at a time for a table. If the constraint being dropped is the parent key in some referential constraint relationship, the constraint between this parent file and any dependent files is also removed.

DROP TABLE and DROP SCHEMA statements also remove any constraints on the table or schema being dropped.

Example: Removing Constraints

The following example removes the primary key over column DEPTNO in table DEPARTMENT. The constraints REPORTS_TO_EXISTS and WORKDEPT_EXISTS defined on tables DEPARTMENT and EMPLOYEE respectively will be removed as well, since the primary key being removed is the parent key in those constraint relationships.

```
ALTER TABLE CORPDATA.EMPLOYEE DROP PRIMARY KEY
```

You can also remove a constraint by name, as in the following example:

```
ALTER TABLE CORPDATA.DEPARTMENT
DROP CONSTRAINT UNIQUE_LNAME_IN_DEPT
```

Inserting into tables with referential constraints

There are some important things to remember when inserting data into tables with referential constraints. If you are inserting data into a parent table with a parent key, SQL does not allow:

- Duplicate values for the parent key
- If the parent key is a primary key, a null value for any column of the primary key

If you are inserting data into a dependent table with foreign keys:

- Each non-null value you insert into a foreign key column must be equal to some value in the corresponding parent key of the parent table.
- If any column in the foreign key is null, the entire foreign key is considered null. If all foreign keys that contain the column are null, the INSERT succeeds (as long as there are no unique index violations).

Example: Inserting data with constraints

Alter the sample application project table (PROJECT) to define two foreign keys:

- A foreign key on the department number (DEPTNO) which references the department table
- A foreign key on the employee number (RESPEMP) which references the employee table.

```
ALTER TABLE CORPDATA.PROJECT ADD CONSTRAINT RESP_DEPT_EXISTS
    FOREIGN KEY (DEPTNO)
    REFERENCES CORPDATA.DEPARTMENT
    ON DELETE RESTRICT
```

```
ALTER TABLE CORPDATA.PROJECT ADD CONSTRAINT RESP_EMP_EXISTS
    FOREIGN KEY (RESPEMP)
    REFERENCES CORPDATA.EMPLOYEE
    ON DELETE RESTRICT
```

Notice that the parent table columns are not specified in the REFERENCES clause. The columns are not required to be specified as long as the referenced table has a primary key or eligible unique key which can be used as the parent key.

Every row inserted into the PROJECT table must have a value of DEPTNO that is equal to some value of DEPTNO in the department table. (The null value is not allowed because DEPTNO in the project table is defined as NOT NULL.) The row must also have a value of RESPEMP that is either equal to some value of EMPNO in the employee table or is null.

The tables with the sample data as they appear in Appendix A, “DB2 UDB for iSeries Sample Tables” conform to these constraints. The following INSERT statement fails because there is no matching DEPTNO value ('A01') in the DEPARTMENT table.

```
INSERT INTO CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3120', 'BENEFITS ADMIN', 'A01', '000010')
```

Likewise, the following INSERT statement would be unsuccessful since there is no EMPNO value of '000011' in the EMPLOYEE table.

```
INSERT INTO CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3130', 'BILLING', 'D21', '000011')
```

The following INSERT statement completes successfully because there is a matching DEPTNO value of 'E01' in the DEPARTMENT table and a matching EMPNO value of '000010' in the EMPLOYEE table.

```
INSERT INTO CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3120', 'BENEFITS ADMIN', 'E01', '000010')
```

Updating tables with referential constraints

If you are updating a *parent* table, you cannot modify a primary key for which dependent rows exist. Changing the key violates referential constraints for dependent tables and leaves some rows without a parent. Furthermore, you cannot give any part of a primary key a null value.

Update Rules

The action taken on dependent tables when an UPDATE is performed on a parent table depends on the update rule specified for the referential constraint. If no update rule was defined for a referential constraint, the UPDATE NO ACTION rule is used.

UPDATE NO ACTION

Specifies that the row in the parent table can be updated if no other row depends on it. If a dependent row exists in the relationship, the UPDATE fails. The check for dependent rows is performed at the end of the statement.

UPDATE RESTRICT

Specifies that the row in the parent table can be updated if no other row depends on it. If a dependent row exists in the relationship, the UPDATE fails. The check for dependent rows is performed immediately.

The subtle difference between the RESTRICT rule and the NO ACTION rule is easiest seen when looking at the interaction of triggers and referential constraints. Triggers can be defined to fire either before or after an operation (an UPDATE statement, in this case). A *before trigger* fires before the UPDATE is performed and therefore before any checking of constraints. An *after trigger* is fired after the UPDATE is performed, and after a constraint rule of RESTRICT (where checking is performed immediately), but before a constraint rule of NO ACTION (where checking is performed at the end of the statement). The triggers and rules would occur in the following order:

1. A *before trigger* would be fired before the UPDATE and before a constraint rule of RESTRICT or NO ACTION.
2. An *after trigger* would be fired after a constraint rule of RESTRICT, but before a NO ACTION rule.

If you are updating a *dependent* table, any non-null foreign key values that you change must match the primary key for each relationship in which the table is a dependent. For example, department numbers in the employee table depend on the department numbers in the department table. You can assign an employee to no department (the null value), but not to a department that does not exist.

If an UPDATE against a table with a referential constraint fails, all changes made during the update operation are undone. For more information about the implications of commitment control and journaling when working with constraints, see “Journaling” on page 310 and “Commitment control” on page 311.

Examples: UPDATE Rules

For example, you cannot update a department number from the department table if it is still responsible for some project, which is described by a dependent row in the project table.

The following UPDATE fails because the PROJECT table has rows that are dependent on DEPARTMENT.DEPTNO having a value of 'D01' (the row targeted

by the WHERE statement). If this UPDATE were allowed, the referential constraint between the PROJECT and DEPARTMENT tables would be broken.

```
UPDATE CORPDATA.DEPARTMENT
  SET DEPTNO = 'D99'
  WHERE DEPTNAME = 'DEVELOPMENT CENTER'
```

The following statement fails because it violates the referential constraint that exists between the primary key DEPTNO in DEPARTMENT and the foreign key DEPTNO in PROJECT:

```
UPDATE CORPDATA.PROJECT
  SET DEPTNO = 'D00'
  WHERE DEPTNO = 'D01';
```

The statement attempts to change all department numbers of D01 to department number D00. Since D00 is not a value of the primary key DEPTNO in DEPARTMENT, the statement fails.

Deleting from tables with referential constraints

If a table has a primary key but no dependents, DELETE operates as it does without referential constraints. The same is true if a table has only foreign keys, but no primary key. If a table has a primary key and dependent tables, DELETE deletes or updates rows according to the delete rules specified. All delete rules of all affected relationships must be satisfied in order for the delete operation to succeed. If a referential constraint is violated, the DELETE fails.

The action to be taken on dependent tables when a DELETE is performed on a parent table depends on the delete rule specified for the referential constraint. If no delete rule was defined, the DELETE NO ACTION rule is used.

DELETE NO ACTION

Specifies that the row in the parent table can be deleted if no other row depends on it. If a dependent row exists in the relationship, the DELETE fails. The check for dependent rows is performed at the end of the statement.

DELETE RESTRICT

Specifies that the row in the parent table can be deleted if no other row depends on it. If a dependent row exists in the relationship, the DELETE fails. The check for dependent rows is performed immediately.

For example, you cannot delete a department from the department table if it is still responsible for some project that is described by a dependent row in the project table.

DELETE CASCADE

Specifies that first the designated rows in the parent table are deleted. Then, the dependent rows are deleted.

For example, you can delete a department by deleting its row in the department table. Deleting the row from the department table also deletes:

- The rows for all departments that report to it
- All departments that report to those departments and so forth.

DELETE SET NULL

Specifies that each nullable column of the foreign key in each dependent row is set to its default value. This means that the column is only set to its

default value if it is a member of a foreign key that references the row being deleted. Only the dependent rows that are immediate descendants are affected.

DELETE SET DEFAULT

Specifies that each column of the foreign key in each dependent row is set to its default value. This means that the column is only set to its default value if it is a member of a foreign key that references the row being deleted. Only the dependent rows that are immediate descendants are affected.

For example, you can delete an employee from the employee table (EMPLOYEE) even if the employee manages some department. In that case, the value of MGRNO for each employee who reported to the manager is set to blanks in the department table (DEPARTMENT). If some other default value was specified on the create of the table, that value is used.

This is due to the REPORTS_TO_EXISTS constraint defined for the department table.

If a descendent table has a delete rule of RESTRICT or NO ACTION and a row is found such that a descendant row cannot be deleted, the entire DELETE fails.

When running this statement with a program, the number of rows deleted is returned in SQLERRD(3) in the SQLCA. This number includes only the number of rows deleted in the table specified in the DELETE statement. It does not include those rows deleted according to the CASCADE rule. SQLERRD(5) in the SQLCA contains the number of rows that were affected by referential constraints in all tables.

The subtle difference between RESTRICT and NO ACTION rules is easiest seen when looking at the interaction of triggers and referential constraints. Triggers can be defined to fire either before or after an operation (a DELETE statement, in this case). A *before trigger* fires before the DELETE is performed and therefore before any checking of constraints. An *after trigger* is fired after the DELETE is performed, and after a constraint rule of RESTRICT (where checking is performed immediately), but before a constraint rule of NO ACTION (where checking is performed at the end of the statement). The triggers and rules would occur in the following order:

1. A *before trigger* would be fired before the DELETE and before a constraint rule of RESTRICT or NO ACTION.
2. An *after trigger* would be fired after a constraint rule of RESTRICT, but before a NO ACTION rule.

Example: DELETE Cascade Rule

Deleting a department from the DEPARTMENT table sets WORKDEPT (in the EMPLOYEE table) to null for every employee assigned to that department.

Consider the following DELETE statement:

```
DELETE FROM CORPDATA.DEPARTMENT
WHERE DEPTNO = 'E11'
```

Given the tables and the data as they appear in Appendix A, “DB2 UDB for iSeries Sample Tables”, one row is deleted from table DEPARTMENT, and table EMPLOYEE is updated to set the value of WORKDEPT to its default wherever the value was 'E11'. A question mark (?) in the sample data below reflects the null value. The results would appear as follows:

Table 22. DEPARTMENT Table. Contents of the table after the DELETE statement is complete.

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00
B01	PLANNING	000020	A00
C01	INFORMATION CENTER	000030	A00
D01	DEVELOPMENT CENTER	?	A00
D11	MANUFACTURING SYSTEMS	000060	D01
D21	ADMINISTRATION SYSTEMS	000070	D01
E01	SUPPORT SERVICES	000050	A00
E21	SOFTWARE SUPPORT	000100	E01
F22	BRANCH OFFICE F2	?	E01
G22	BRANCH OFFICE G2	?	E01
H22	BRANCH OFFICE H2	?	E01
I22	BRANCH OFFICE I2	?	E01
J22	BRANCH OFFICE J2	?	E01

Note that there were no cascaded deletes in the DEPARTMENT table because no department reported to department 'E11'.

Below are snapshots of one affected portion of the EMPLOYEE table before and after the DELETE statement is completed.

Table 23. Partial EMPLOYEE Table. Partial contents before the DELETE statement.

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05
000250	DANIEL	S	SMITH	D21	0961	1960-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	E11	0997	1967-03-24
000290	JOHN	R	PARKER	E11	4502	1980-05-30
000300	PHILIP	X	SMITH	E11	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23
000340	JASON	R	GOUNOT	E21	5696	1947-05-05

Table 24. Partial EMPLOYEE Table. Partial contents after the DELETE statement.

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05
000250	DANIEL	S	SMITH	D21	0961	1960-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11

Table 24. Partial EMPLOYEE Table (continued). Partial contents after the DELETE statement.

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	?	0997	1967-03-24
000290	JOHN	R	PARKER	?	4502	1980-05-30
000300	PHILIP	X	SMITH	?	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	?	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23
000340	JASON	R	GOUNOT	E21	5696	1947-05-05

Check pending

Referential constraints and check constraints can be in a state known as check pending, where potential violations of the constraint exist. For referential constraints, a violation occurs when potential mismatches exist between parent and foreign keys. For check constraints, a violation occurs when potential values exist in columns which are limited by the check constraint. When the system determines that the constraint may have been violated (such as after a restore operation), the constraint is marked as check pending. When this happens, restrictions are placed on the use of tables involved in the constraint. For referential constraints, the following restrictions apply:

- No input or output operations are allowed on the dependent file.
- Only read and insert operations are allowed on the parent file.

When a check constraint is in check pending, the following restrictions apply:

- Read operations are not allowed on the file.
- Inserts and updates are allowed and the constraint is enforced.

To get a constraint out of check pending, you must:

1. Disable the relationship with the Change Physical File Constraint (CHGPFCST) CL command.
2. Correct the key (foreign, parent, or both) data for referential constraints or column data for check constraints.
3. Enable the constraint again with the CHGPFCST CL command.

You can identify the rows that are in violation of the constraint with the Display Check Pending Constraint (DSPCPCST) CL command.

For more information about working with tables in check pending, see the Database Programming book.

WITH CHECK OPTION on a View

WITH CHECK OPTION is an optional clause on the CREATE VIEW statement that specifies the level of checking to be done when inserting or updating data through a view. If the option is specified, every row that is inserted or updated through the view must conform to the definition of that view.

WITH CHECK OPTION cannot be specified if the view is read-only. The definition of the view must not include a subquery.

If the view is created without a WITH CHECK OPTION clause, insert and update operations that are performed on the view are not checked for conformance to the view definition. Some checking might still occur if the view is directly or indirectly dependent on another view that includes WITH CHECK OPTION. Because the definition of the view is not used, rows might be inserted or updated through the view that do not conform to the definition of the view. This means that the rows could not be selected again using the view.

The checking can either be “WITH CASCADED CHECK OPTION” or “WITH LOCAL CHECK OPTION” on page 146. See the CREATE VIEW topic in the SQL Reference book for additional discussion of WITH CHECK OPTION.

WITH CASCADED CHECK OPTION

The WITH CASCADED CHECK OPTION specifies that every row that is inserted or updated through the view must conform to the definition of the view. In addition, the search conditions of all dependent views are checked when a row is inserted or updated. If a row does not conform to the definition of the view, that row cannot be retrieved using the view.

For example, consider the following updateable view:

```
CREATE VIEW V1 AS SELECT COL1
FROM T1 WHERE COL1 > 10
```

Because no WITH CHECK OPTION is specified, the following INSERT statement is successful even though the value being inserted does not meet the search condition of the view.

```
INSERT INTO V1 VALUES (5)
```

Create another view over V1, specifying the WITH CASCADED CHECK OPTION:

```
CREATE VIEW V2 AS SELECT COL1
FROM V1 WITH CASCADED CHECK OPTION
```

The following INSERT statement fails because it would produce a row that does not conform to the definition of V2:

```
INSERT INTO V2 VALUES (5)
```

Consider one more view created over V2:

```
CREATE VIEW V3 AS SELECT COL1
FROM V2 WHERE COL1 < 100
```

The following INSERT statement fails only because V3 is dependent on V2, and V2 has a WITH CASCADED CHECK OPTION.

```
INSERT INTO V3 VALUES (5)
```

However, the following INSERT statement is successful because it conforms to the definition of V2. Because V3 does not have a WITH CASCADED CHECK OPTION, it does not matter that the statement does not conform to the definition of V3.

```
INSERT INTO V3 VALUES (200)
```

WITH LOCAL CHECK OPTION

WITH LOCAL CHECK OPTION is identical to WITH CASCADED CHECK OPTION except that you can update a row so that it no longer can be retrieved through the view. This can only happen when the view is directly or indirectly dependent on a view that was defined with no WITH CHECK OPTION clause.

For example, consider the same updateable view used in the previous example:

```
CREATE VIEW V1 AS SELECT COL1
FROM T1 WHERE COL1 > 10
```

Create second view over V1, this time specifying WITH LOCAL CHECK OPTION:

```
CREATE VIEW V2 AS SELECT COL1
FROM V1 WITH LOCAL CHECK OPTION
```

The same INSERT that failed in the previous CASCADED CHECK OPTION example would succeed now because V2 does not have any search conditions, and the search conditions of V1 do not need to be checked since V1 does not specify a check option.

```
INSERT INTO V2 VALUES (5)
```

If we again consider one more view created over V2:

```
CREATE VIEW V3 AS SELECT COL1
FROM V2 WHERE COL1 < 100
```

The following INSERT is successful again because the search condition on V1 is not checked due to the WITH LOCAL CHECK OPTION on V2, versus the WITH CASCADED CHECK OPTION in the previous example.

```
INSERT INTO V3 VALUES (5)
```

The difference between LOCAL and CASCADED CHECK OPTION lies in how many of the dependent views' search conditions are checked when a row is inserted or updated.

- WITH LOCAL CHECK OPTION specifies that the search conditions of only those dependent views that have the WITH LOCAL CHECK OPTION or WITH CASCADED CHECK OPTION are checked when a row is inserted or updated.
- WITH CASCADED CHECK OPTION specifies that the search conditions of all dependent views are checked when a row is inserted or updated.

Example: Cascaded check option

Use the following table and views:

```
CREATE TABLE T1 (COL1 CHAR(10))

CREATE VIEW V1 AS SELECT COL1
FROM T1 WHERE COL1 LIKE 'A%'

CREATE VIEW V2 AS SELECT COL1
FROM V1 WHERE COL1 LIKE '%Z'
WITH LOCAL CHECK OPTION

CREATE VIEW V3 AS SELECT COL1
FROM V2 WHERE COL1 LIKE 'AB%'

CREATE VIEW V4 AS SELECT COL1
FROM V3 WHERE COL1 LIKE '%YZ'
WITH CASCADED CHECK OPTION

CREATE VIEW V5 AS SELECT COL1
FROM V4 WHERE COL1 LIKE 'ABC%'
```


Different search conditions are going to be checked depending on which view is being operated on with an INSERT or UPDATE.

- If V1 is operated on, no conditions are checked because V1 does not have a WITH CHECK OPTION specified.
- If V2 is operated on,
 - COL1 must end in the letter Z, but it doesn't have to start with the letter A. This is because the check option is LOCAL, and view V1 does not have a check option specified.
- If V3 is operated on,
 - COL1 must end in the letter Z, but it does not have to start with the letter A. V3 does not have a check option specified, so its own search condition must not be met. However, the search condition for V2 must be checked since V3 is defined on V2, and V2 has a check option.
- If V4 is operated on,
 - COL1 must start with 'AB', and must end with 'YZ'. Because V4 has the WITH CASCADED CHECK OPTION specified, every search condition for every view on which V4 is dependent must be checked.
- If V5 is operated on,
 - COL1 must start with 'AB', but not necessarily 'ABC'. This is because V5 does not specify a check option, so its own search condition does not need to be checked. However, because V5 is defined on V4, and V4 had a cascaded check option, every search condition for V4, V3, V2, and V1 must be checked. That is, COL1 must start with 'AB' and end with 'YZ'.

If V5 were created WITH LOCAL CHECK OPTION, operating on V5 would mean that COL1 must start with 'ABC' and end with 'YZ'. The LOCAL CHECK OPTION adds the additional requirement that the third character must be a 'C'.

DB2 UDB for iSeries Trigger support

A *trigger* is a set of actions that are run automatically when a specified change operation is performed on a specified table. The change operation can be an SQL INSERT, UPDATE, or DELETE statement, or an insert, update, or delete high level language statement in an application program. Triggers are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail.

Triggers can be defined in two different ways:

- "SQL triggers" on page 148
- "External triggers" on page 152

For an external trigger, the CRTPFTRG CL command is used. The program containing the set of trigger actions can be defined in any supported high level language. External triggers can be insert, update, delete, or read triggers.

For an SQL trigger, the CREATE TRIGGER statement is used. The trigger program is defined entirely using SQL. SQL triggers can be insert, update, or delete triggers.

Once a trigger is associated with a table, the trigger support calls the trigger program whenever a change operation is initiated against the table, or any logical file or view created over the table. SQL triggers and external triggers can be defined for the same table. Up to 200 triggers can be defined for a single table.

Each change operation can call a trigger before or after the change operation occurs. Additionally, you can add a *read* trigger that is called every time the table is accessed. Thus, a table can be associated with many types of triggers.

- Before delete trigger
- Before insert trigger
- Before update trigger
- After delete trigger
- After insert trigger
- After update trigger
- Read only trigger (external trigger only)

See the "Triggering automatic events in the your database" chapter in the Database Programming book for information about trigger limits, including how many triggers may be defined for an SQL table and the maximum trigger nesting level, and for recommendations and precautions when coding a trigger.

SQL triggers

The SQL CREATE TRIGGER statement provides a way for the database management system to actively control, monitor, and manage a group of tables whenever an insert, update, or delete operation is performed. The statements specified in the SQL trigger are executed each time an SQL insert, update, or delete operation is performed. An SQL trigger may call stored procedures or user-defined functions to perform additional processing when the trigger is executed.

Unlike stored procedures, an SQL trigger cannot be directly called from an application. Instead, an SQL trigger is invoked by the database management system upon the execution of a triggering insert, update, or delete operation. The definition of the SQL trigger is stored in the database management system and is invoked by the database management system, when the SQL table, that the trigger is defined on, is modified.

For more information about creating SQL triggers, see "Creating an SQL trigger".

For complete details about using the CREATE TRIGGER statement, see the CREATE TRIGGER statement in the SQL Reference topic.

Creating an SQL trigger

An SQL trigger can be created by specifying the CREATE TRIGGER SQL statement. The statements in the routine-body of the SQL trigger are transformed by SQL into a program (*PGM) object. The program is created in the schema specified by the trigger name qualifier. The specified trigger is registered in the SYSTRIGGERS, SYSTRIGDEP, SYSTRIGCOL, and SYSTRIGUPD SQL Catalogs. See the "SQL procedures, functions, and triggers" chapter in the SQL Reference for additional information about how to use variable control statements in an SQL trigger and for information about how to debug an SQL trigger at the SQL statement level.

For some examples and considerations of creating SQL triggers, see:

- "BEFORE SQL triggers" on page 149
- "AFTER SQL triggers" on page 150
- "Handlers in SQL triggers" on page 151
- "SQL trigger transition tables" on page 152

BEFORE SQL triggers

BEFORE triggers may not modify tables, but they can be used to verify input column values, and also to modify column values that are inserted or updated in a table. In the following example, the trigger is used to set the fiscal quarter for the corporation prior to inserting the row into the target table.

```
CREATE TABLE TransactionTable (DateOfTransaction DATE, FiscalQuarter SMALLINT)
```

```
CREATE TRIGGER TransactionBeforeTrigger BEFORE INSERT ON TransactionTable  
REFERENCING NEW AS new_row  
FOR EACH ROW MODE DB2ROW  
BEGIN  
  DECLARE newmonth SMALLINT;  
SET newmonth = MONTH(new_row.DateOfTransaction);  
  IF newmonth < 4 THEN  
    SET new_row.FiscalQuarter=3;  
  ELSEIF newmonth < 7 THEN  
    SET new_row.FiscalQuarter=4;  
  ELSEIF newmonth < 10 THEN  
    SET new_row.FiscalQuarter=1;  
  ELSE  
    SET new_row.FiscalQuarter=2;  
  END IF;  
END
```

For the SQL insert statement below, the "FiscalQuarter" column would be set to 2, if the current date is November 14, 2000.

```
INSERT INTO TransactionTable(DateOfTransaction)  
  VALUES(CURRENT DATE)
```

SQL triggers have access to and can use User-defined Distinct Types (UDTs) and stored procedures. In the following example, the SQL trigger calls a stored procedure to execute some predefined business logic, in this case, to set a column to a predefined value for the business.

```
CREATE DISTINCT TYPE enginesize AS DECIMAL(5,2) WITH COMPARISONS
```

```
CREATE DISTINCT TYPE engineclass AS VARCHAR(25) WITH COMPARISONS
```

```
CREATE PROCEDURE SetEngineClass(IN SizeInLiters enginesize,  
                                OUT CLASS engineclass)
```

```
LANGUAGE SQL CONTAINS SQL  
BEGIN  
  IF SizeInLiters<2.0 THEN  
    SET CLASS = 'Mouse';  
  ELSEIF SizeInLiters<3.1 THEN  
    SET CLASS = 'Economy Class';  
  ELSEIF SizeInLiters<4.0 THEN  
    SET CLASS = 'Most Common Class';  
  ELSEIF SizeInLiters<4.6 THEN  
    SET CLASS = 'Getting Expensive';  
  ELSE  
    SET CLASS = 'Stop Often for Fillups';  
  END IF;  
END
```

```
CREATE TABLE EngineRatings (VariousSizes enginesize, ClassRating engineclass)
```

```
CREATE TRIGGER SetEngineClassTrigger BEFORE INSERT ON EngineRatings  
REFERENCING NEW AS new_row  
FOR EACH ROW MODE DB2ROW  
  CALL SetEngineClass(new_row.VariousSizes, new_row.ClassRating)
```

For the SQL insert statement below, the "ClassRating" column would be set to "Economy Class", if the "VariousSizes" column has the value of 3.0.

```
INSERT INTO EngineRatings(VariousSizes) VALUES(3.0)
```

SQL requires all tables, user-defined functions, procedures and user-defined types to exist prior to creating an SQL trigger. In the examples above, all of the tables, stored procedures, and user-defined types are defined before the trigger is created.

AFTER SQL triggers

The WHEN condition can be used in an SQL trigger to specify a condition. If the condition evaluates to true, then the SQL statements in the SQL trigger routine body are executed. If the condition evaluates to false, the SQL statements in the SQL trigger routine body are not executed, and control is returned to the database system. In the following example, a query is evaluated to determine if the statements in the trigger routine body should be run when the trigger is activated.

```
CREATE TABLE TodaysRecords(TodaysMaxBarometricPressure FLOAT,  
    TodaysMinBarometricPressure FLOAT)
```

```
CREATE TABLE OurCitysRecords(RecordMaxBarometricPressure FLOAT,  
    RecordMinBarometricPressure FLOAT)
```

```
CREATE TRIGGER UpdateMaxPressureTrigger  
AFTER UPDATE OF TodaysMaxBarometricPressure ON TodaysRecords  
REFERENCING NEW AS new_row  
FOR EACH ROW MODE DB2ROW  
WHEN (new_row.TodaysMaxBarometricPressure >  
    (SELECT MAX(RecordMaxBarometricPressure) FROM  
        OurCitysRecords))  
UPDATE OurCitysRecords  
    SET RecordMaxBarometricPressure =  
        new_row.TodaysMaxBarometricPressure
```

```
CREATE TRIGGER UpdateMinPressureTrigger  
AFTER UPDATE OF TodaysMinBarometricPressure  
ON TodaysRecords  
REFERENCING NEW AS new_row  
FOR EACH ROW MODE DB2ROW  
WHEN(new_row.TodaysMinBarometricPressure <  
    (SELECT MIN(RecordMinBarometricPressure) FROM  
        OurCitysRecords))  
UPDATE OurCitysRecords  
    SET RecordMinBarometricPressure =  
        new_row.TodaysMinBarometricPressure
```

First the current values are initialized for the tables.

```
INSERT INTO TodaysRecords VALUES(0.0,0.0)  
INSERT INTO OurCitysRecords VALUES(0.0,0.0)
```

For the SQL update statement below, the RecordMaxBarometricPressure in OurCitysRecords is updated by the UpdateMaxPressureTrigger.

```
UPDATE TodaysRecords SET TodaysMaxBarometricPressure = 29.95
```

But tomorrow, if the TodaysMaxBarometricPressure is only 29.91, then the RecordMaxBarometricPressure is not updated.

```
UPDATE TodaysRecords SET TodaysMaxBarometricPressure = 29.91
```

SQL allows the definition of multiple triggers for a single triggering action. In the previous example, there are two AFTER UPDATE triggers:

UpdateMaxPressureTrigger and UpdateMinPressureTrigger. These triggers are only activated when specific columns of the table TodaysRecords are updated.

AFTER triggers may modify tables. In the example above, an UPDATE operation is applied to a second table. Note that recursive insert and update operations should be avoided. The database management system terminates the operation if the maximum trigger nesting level is reached. You can avoid recursion by adding conditional logic so that the insert or update operation is exited before the maximum nesting level is reached. The same situation needs to be avoided in a network of triggers that recursively cascade through the network of triggers.

Handlers in SQL triggers

A handler in an SQL trigger gives the SQL trigger the ability to recover from an error or log information about an error that has occurred while executing the SQL statements in the trigger routine body.

In the following example, there are two handlers defined: one to handle the overflow condition and a second handler to handle SQL exceptions.

```
CREATE TABLE ExcessInventory(Description VARCHAR(50), ItemWeight SMALLINT)
```

```
CREATE TABLE YearToDateTotals(TotalWeight SMALLINT)
```

```
CREATE TABLE FailureLog(Item VARCHAR(50), ErrorMessage VARCHAR(50), ErrorCode INT)
```

```
CREATE TRIGGER InventoryDeleteTrigger
AFTER DELETE ON ExcessInventory
REFERENCING OLD AS old_row
FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE sqlcode INT;
  DECLARE invalid_number condition FOR '22003';
  DECLARE exit handler FOR invalid_number
  INSERT INTO FailureLog VALUES(old_row.Description,
    'Overflow occurred in YearToDateTotals', sqlcode);
  DECLARE exit handler FOR sqlexception
  INSERT INTO FailureLog VALUES(old_row.Description,
    'SQL Error occurred in InventoryDeleteTrigger', sqlcode);
  UPDATE YearToDateTotals SET TotalWeight=TotalWeight +
    old_row.itemWeight;
END
```

First, the current values for the tables are initialized.

```
INSERT INTO ExcessInventory VALUES('Desks',32500)
INSERT INTO ExcessInventory VALUES('Chairs',500)
INSERT INTO YearToDateTotals VALUES(0)
```

When the first SQL delete statement below is executed, the ItemWeight for the item "Desks" is added to the column total for TotalWeight in the table YearToDateTotals. When the second SQL delete statement is executed, an overflow occurs when the ItemWeight for the item "Chairs" is added to the column total for TotalWeight, as the column only handles values up to 32767. When the overflow occurs, the invalid_number exit handler is executed and a row is written to the FailureLog table. The sqlexception exit handler would be run, for example, if the YearToDateTotals table was deleted by accident. In this example, the handlers are used to write a log so that the problem can be diagnosed at a later time.

```
DELETE FROM ExcessInventory WHERE Description='Desks'
DELETE FROM ExcessInventory WHERE Description='Chairs'
```

SQL trigger transition tables

An SQL trigger may need to refer to all of the affected rows for an SQL insert, update, or delete operation. This is true, for example, if the trigger needs to apply aggregate functions, such as MIN or MAX, to a specific column of the affected rows. The OLD_TABLE and NEW_TABLE transition tables can be used for this purpose. In the following example, the trigger applies the aggregate function MAX to all of the affected rows of the table StudentProfiles.

```
CREATE TABLE StudentProfiles(StudentsName VARCHAR(125),
    StudentsYearInSchool SMALLINT, StudentsGPA DECIMAL(5,2))

CREATE TABLE CollegeBoundStudentsProfile
    (YearInSchoolMin SMALLINT, YearInSchoolMax SMALLINT, StudentGPAMin
    DECIMAL(5,2), StudentGPAMax DECIMAL(5,2))

CREATE TRIGGER UpdateCollegeBoundStudentsProfileTrigger
AFTER UPDATE ON StudentProfiles
REFERENCING NEW_TABLE AS ntable
FOR EACH STATEMENT MODE DB2SQL
BEGIN
    DECLARE maxStudentYearInSchool SMALLINT;
    SET maxStudentYearInSchool =
        (SELECT MAX(StudentsYearInSchool) FROM ntable);
    IF maxStudentYearInSchool >
        (SELECT MAX (YearInSchoolMax) FROM
            CollegeBoundStudentsProfile) THEN
        UPDATE CollegeBoundStudentsProfile SET YearInSchoolMax =
            maxStudentYearInSchool;
    END IF;
END
```

In the preceding example, the trigger is executed a single time following the execution of a triggering update statement because it is defined as a FOR EACH STATEMENT trigger. You will need to consider the processing overhead required by the database management system for populating the transition tables when you define a trigger that references transition tables.

External triggers

For an external trigger, the program containing the set of trigger actions can be defined in any supported high level language that creates a *PGM object. The trigger program can have SQL embedded in it. To define an external trigger, you must create a trigger program and add it to a table using the ADDPFTRG CL command or you can add it using iSeries Navigator. To add a trigger to a table, you must:

- Identify the table
- Identify the kind of operation
- Identify the program that performs the desired actions.

For an example of an external trigger, see "External trigger example program".

External trigger example program

A sample external trigger program follows. It is written in ILE C, with embedded SQL.

See "Triggering automatic events in your database" chapter in the Database Programming book for a full discussion and more examples of external trigger usage in DB2 UDB for iSeries.

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

```

#include "string.h"
#include "stdlib.h"
#include "stdio.h"
#include <recio.h>
#include <xxcvt.h>
#include "qsysinc/h/trgbuf"      /* Trigger input parameter      */
#include "libl/csrc/msghand1"    /* User defined message handler  */
/*****
/* This is a trigger program which is called whenever there is an
/* update to the EMPLOYEE table. If the employee's commission is
/* greater than the maximum commission, this trigger program will
/* increase the employee's salary by 1.04 percent and insert into
/* the RAISE table.
/*
/* The EMPLOYEE record information is passed from the input parameter*/
/* to this trigger program.
*****/

Qdb_Trigger_Buffer_t *hstruct;
char *datap;

/*****
/* Structure of the EMPLOYEE record which is used to
/* store the old or the new record that is passed to
/* this trigger program.
/*
/* Note : You must ensure that all the numeric fields
/* are aligned at 4 byte boundary in C.
/* Used either Packed struct or filler to reach
/* the byte boundary alignment.
*****/

_Packed struct rec{
    char empn[6];
    _Packed struct { short fstlen ;
                    char fstnam[12];
                    } fstname;
    char minit[1];
    _Packed struct { short lstlen;
                    char lstnam[15];
                    } lstname;
    char dept[3];
    char phone[4];
    char hdate[10];
    char jobn[8];
    short edclvl;
    char sex1[1];
    char bdate[10];
    decimal(9,2) salary1;
    decimal(9,2) bonus1;
    decimal(9,2) comm1;
    } oldbuf, newbuf;
EXEC SQL INCLUDE SQLCA;

```

Figure 2. Sample Trigger Program (Part 1 of 5)

```

main(int argc, char **argv)
{
int i;
int obufoff;           /* old buffer offset      */
int nulloff;          /* old null byte map offset */
int nbufoff;          /* new buffer offset      */
int nul2off;          /* new null byte map offset */
short work_days = 253; /* work days during in one year */
decimal(9,2) commission = 2000.00; /* cutoff to qualify for */
decimal(9,2) percentage = 1.04; /* raised salary as percentage */
char raise_date[12] = "1982-06-01"; /* effective raise date */

struct {
    char empno[6];
    char name[30];
    decimal(9,2) salary;
    decimal(9,2) new_salary;
    } rpt1;

    /* Start to monitor any exception. */
    /* ***** */

    _FEEDBACK fc;
    _HDLR_ENTRY hdlr = main_handler;
    /* ***** */
    /* Make the exception handler active. */
    /* ***** */
    CEEHDLR(&hdlr, NULL, &fc);
    /* ***** */
    /* Ensure exception handler OK */
    /* ***** */

    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register exception handler.\n");
        exit(99);
    };

    /* ***** */
    /* Move the data from the trigger buffer to the local */
    /* structure for reference. */
    /* ***** */

    hstruct = (Qdb_Trigger_Buffer_t *)argv[1];
    datapt = (char *) hstruct;

    obufoff = hstruct ->Old_Record_Offset; /* old buffer */
    memcpy(&oldbuf,datapt+obufoff,; hstruct->Old_Record_Len);

    nbufoff = hstruct ->New_Record_Offset; /* new buffer */
    memcpy(&newbuf,datapt+nbufoff,; hstruct->New_Record_Len);

```

Figure 2. Sample Trigger Program (Part 2 of 5)


```

EXEC SQL WHENEVER SQLERROR GO TO ERR_EXIT;

/*****
/* Set the transaction isolation level to the same as */
/* the application based on the input parameter in the */
/* trigger buffer. */
*****/

if(strcmp(hstruct->Commit_Lock_Level,"0") == 0)
    EXEC SQL SET TRANSACTION ISOLATION LEVEL NONE;
else{
    if(strcmp(hstruct->Commit_Lock_Level,"1") == 0)
        EXEC SQL SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED, READ
            WRITE;
    else {
        if(strcmp(hstruct->Commit_Lock_Level,"2") == 0)
            EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
        else
            if(strcmp(hstruct->Commit_Lock_Level,"3") == 0)
                EXEC SQL SET TRANSACTION ISOLATION LEVEL ALL;
    }
}

/*****
/* If the employee's commission is greater than maximum */
/* commission, then increase the employee's salary */
/* by 1.04 percent and insert into the RAISE table. */
*****/

if (newbuf.comm1 >= commission)
{
    EXEC SQL SELECT EMPNO, EMPNAME, SALARY
        INTO :rpt1.empno, :rpt1.name, :rpt1.salary
        FROM TRGPERF/EMP_ACT
        WHERE EMP_ACT.EMPNO=:newbuf.empn ;

    if (sqlca.sqlcode == 0) then
    {
        rpt1.new_salary = salary * percentage;
        EXEC SQL INSERT INTO TRGPERF/RAISE VALUES(:rpt1);
    }
    goto finished;
}
err_exit:
    exit(1);

/* All done */
finished:
    return;
} /* end of main line */

```

Figure 2. Sample Trigger Program (Part 3 of 5)

```

/*****
/* INCLUDE NAME : MSGHAND1 */
/* */
/* DESCRIPTION : Message handler to signal an exception to */
/* the application to inform that an */
/* error occurred in the trigger program. */
/* */
/* NOTE : This message handler is a user defined routine. */
/* */
*****/
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#include <leawi.h>

#pragma linkage (QMHSNDPM, OS)
void QMHSNDPM(char *, /* Message identifier */
void *, /* Qualified message file name */
void *, /* Message data or text */
int, /* Length of message data or text */
char *, /* Message type */
char *, /* Call message queue */
int, /* Call stack counter */
void *, /* Message key */
void *, /* Error code */
...); /* Optionals:
length of call message queue
name
Call stack entry qualification
display external messages
screen wait time */
/*****
/***** This is the start of the exception handler function. */
/*****
void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc,
_FEEDBACK *new)
{
/* Initialize variables for call to
/* QMHSNDPM.
/* User must create a message file and
/* define a message ID to match the
/* following data.
*****/
char message_id[7] = "TRG9999";
char message_file[20] = "MSGF LIB1 ";
char message_data[50] = "Trigger error ";
int message_len = 30;
char message_type[10] = "*ESCAPE ";
char message_q[10] = "_C_pep ";
int pgm_stack_cnt = 1;
char message_key[4];

```

Figure 2. Sample Trigger Program (Part 4 of 5)

```

/******
/* Declare error code structure for      */
/* QMHSNDPM.                            */
/******
struct error_code {
    int bytes_provided;
    int bytes_available;
    char message_id[7];
} error_code;

error_code.bytes_provided = 15;
/******
/* Set the error handler to resume and  */
/* mark the last escape message as     */
/* handled.                             */
/******
*rc = CEE_HDLR_RESUME;
/******
/* Send my own *ESCAPE message.        */
/******
QMHSNDPM(message_id,
          &message_file,
          &message_data,
          message_len,
          message_type,
          message_q,
          pgm_stack_cnt,
          &message_key,
          &error_code );
/******
/* Check that the call to QMHSNDPM     */
/* finished correctly.                  */
/******
if (error_code.bytes_available != 0)
    {
        printf("Error in QMHOVPM : %s\n", error_code.message_id);
    }
}

```

Figure 2. Sample Trigger Program (Part 5 of 5)

Chapter 11. Stored Procedures

A *procedure* (often called a stored procedure) is a program that can be called to perform operations that can include both host language statements and SQL statements. Procedures in SQL provide the same benefits as procedures in a host language.

DB2 SQL for iSeries stored procedure support provides a way for an SQL application to define and then invoke a procedure through SQL statements. Stored procedures can be used in both distributed and non-distributed DB2 SQL for iSeries applications. One of the big advantages in using stored procedures is that for distributed applications, the execution of one CALL statement on the application requester, or client, can perform any amount of work on the application server.

You may define a procedure as either an SQL procedure or an external procedure. An external procedure can be any supported high level language program (except System/36* programs and procedures) or a REXX procedure. The procedure does not need to contain SQL statements, but it may contain SQL statements. An SQL procedure is defined entirely in SQL, and can contain SQL statements that include SQL control statements.

Coding stored procedures requires that the user understand the following:

- Stored procedure definition through the CREATE PROCEDURE statement
- Stored procedure invocation through the CALL statement
- Parameter passing conventions
- Methods for returning a completion status to the program invoking the procedure.

You may define stored procedures by using the CREATE PROCEDURE statement. The CREATE PROCEDURE statement adds procedure and parameter definitions to the catalog tables SYSROUTINES and SYSPARMS. These definitions are then accessible by any SQL CALL statement on the system.

To create an external procedure or an SQL procedure, you can use the SQL CREATE PROCEDURE statement. Or, you can define a procedure using iSeries Navigator.

The following sections describe the SQL statements used to define and invoke the stored procedure, information on passing parameters to the stored procedure, and examples of stored procedure usage.

- “Defining an external procedure” on page 160
- “Defining an SQL procedure” on page 161
- “Debugging a stored procedure” on page 166
- “Invoking a stored procedure” on page 167
- “Parameter passing conventions for stored procedures and UDFs” on page 171
- “Indicator variables and stored procedures” on page 176
- “Returning a completion status to the calling program” on page 178
- “Examples of CALL statements” on page 179

For a description of stored procedures coded in Java, see Java SQL Routines in the IBM Developer Kit for Java topic.

For information about using stored procedures with DRDA, see “DRDA stored procedure considerations” on page 351.

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Defining an external procedure

The CREATE PROCEDURE statement for an external procedure:

- Names the procedure
- Defines the parameters and their attributes
- Gives other information about the procedure which will the system uses when it calls the procedure.

Consider the following example:

```
CREATE PROCEDURE P1
  (INOUT PARM1 CHAR(10))
  EXTERNAL NAME MYLIB.PROC1
  LANGUAGE C
  GENERAL WITH NULLS;
```

This CREATE PROCEDURE statement:

- Names the procedure P1
- Defines one parameter which is used both as an input parameter and an output parameter. The parameter is a character field of length ten. Parameters can be defined to be type IN, OUT, or INOUT. The parameter type determines when the values for the parameters get passed to and from the procedure.
- Defines the name of the program which corresponds to the procedure, which is PROC1 in MYLIB. MYLIB.PROC1 is the program which is called when the procedure is invoked on a CALL statement.
- Indicates that the procedure P1 (program MYLIB.PROC1) is written in C. The language is important since it impacts the types of parameters that can be passed. It also affects how the parameters are passed to the procedure (for example, for ILE C procedures, a NUL-terminator is passed on character, graphic, date, time, and timestamp parameters).
- Defines the CALL type to be GENERAL WITH NULLS. This indicates that the parameter for the procedure can possibly contain the NULL value, and therefore would like an additional argument passed to the procedure on the CALL statement. The additional argument is an array of N short integers, where N is the number of parameters that are declared in the CREATE PROCEDURE statement. In this example, the array contains only one element since there is only parameter.

It is important to note that it is not necessary to define a procedure in order to call it. However, if no procedure definition is found, either from a prior CREATE PROCEDURE or from a DECLARE PROCEDURE in this program, certain restrictions and assumptions are made when the procedure is invoked on the CALL statement. For example, the NULL indicator argument cannot be passed. See “Using Embedded CALL Statement where no procedure definition exists” on page 168 for an example of a CALL statement without a corresponding procedure definition.

Defining an SQL procedure

The CREATE PROCEDURE statement for SQL procedures:

- Names the procedure
- Defines the parameters and their attributes
- Provides other information about the procedure which will be used when the procedure is called
- Defines the procedure body. The procedure body is the executable part of the procedure and is a single SQL statement.

Consider the following simple example that takes as input an employee number and a rate and updates the employee's salary:

```
CREATE PROCEDURE UPDATE_SALARY_1
(IN EMPLOYEE_NUMBER CHAR(10),
 IN RATE DECIMAL(6,2))
LANGUAGE SQL MODIFIES SQL DATA
UPDATE CORPDATA.EMPLOYEE
SET SALARY = SALARY * RATE
WHERE EMPNO = EMPLOYEE_NUMBER;
```

This CREATE PROCEDURE statement:

- Names the procedure UPDATE_SALARY_1.
- Defines parameter EMPLOYEE_NUMBER which is an input parameter and is a character data type of length 6 and parameter RATE which is an input parameter and is a decimal data type.
- Indicates the procedure is an SQL procedure that modifies SQL data.
- Defines the procedure body as a single UPDATE statement. When the procedure is called, the UPDATE statement is executed using the values passed for EMPLOYEE_NUMBER and RATE.

Instead of a single UPDATE statement, logic can be added to the SQL procedure using SQL control statements. SQL control statements consist of the following:

- an assignment statement
- a CALL statement
- a CASE statement
- a compound statement
- a FOR statement
- a GET DIAGNOSTICS statement
- a GOTO statement
- an IF statement
- an ITERATE statement
- a LEAVE statement
- a LOOP statement
- a REPEAT statement
- a RESIGNAL statement
- a RETURN statement
- a SIGNAL statement
- a WHILE statement

The following example takes as input the employee number and a rating that was received on the last evaluation. The procedure uses a CASE statement to determine the appropriate increase and bonus for the update:

```

CREATE PROCEDURE UPDATE_SALARY_2
  (IN EMPLOYEE_NUMBER CHAR(6),
   IN RATING INT)
  LANGUAGE SQL MODIFIES SQL DATA
  CASE RATING
    WHEN 1 THEN
      UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY * 1.10,
          BONUS = 1000
        WHERE EMPNO = EMPLOYEE_NUMBER;
    WHEN 2 THEN
      UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY * 1.05,
          BONUS = 500
        WHERE EMPNO = EMPLOYEE_NUMBER;
    ELSE
      UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY * 1.03,
          BONUS = 0
        WHERE EMPNO = EMPLOYEE_NUMBER;
  END CASE;

```

This CREATE PROCEDURE statement:

- Names the procedure UPDATE_SALARY_2.
- Defines parameter EMPLOYEE_NUMBER which is an input parameter and is a character data type of length 6 and parameter RATING which is an input parameter and is an integer data type.
- Indicates the procedure is an SQL procedure that modifies SQL data.
- Defines the procedure body. When the procedure is called, input parameter RATING is checked and the appropriate update statement is executed.

Multiple statements can be added to a procedure body by adding a compound statement. Within a compound statement, any number of SQL statements can be specified. In addition, SQL variables, cursors, and handlers can be declared.

The following example takes as input the department number. It returns the total salary of all the employees in that department and the number of employees in that department who get a bonus.

```

CREATE PROCEDURE RETURN_DEPT_SALARY
  (IN DEPT_NUMBER CHAR(3),
   OUT DEPT_SALARY DECIMAL(15,2),
   OUT DEPT_BONUS_CNT INT)
  LANGUAGE SQL READS SQL DATA
  P1: BEGIN
    DECLARE EMPLOYEE_SALARY DECIMAL(9,2);
    DECLARE EMPLOYEE_BONUS DECIMAL(9,2);
    DECLARE TOTAL_SALARY DECIMAL(15,2)DEFAULT 0
    DECLARE BONUS_CNT INT DEFAULT 0;
    DECLARE END_TABLE INT DEFAULT 0;
    DECLARE C1 CURSOR FOR
      SELECT SALARY, BONUS FROM CORPDATA.EMPLOYEE
        WHERE WORKDEPT = DEPT_NUMBER;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
      SET END_TABLE = 1;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
      SET DEPT_SALARY = NULL;
    OPEN C1;
    FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;

```



```

WHILE END_TABLE = 0 DO
  SET TOTAL_SALARY = TOTAL_SALARY + EMPLOYEE_SALARY + EMPLOYEE_BONUS;
  IF EMPLOYEE_BONUS > 0 THEN
    SET BONUS_CNT = BONUS_CNT + 1;
  END IF;
  FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
END WHILE;
CLOSE C1;
SET DEPT_SALARY = TOTAL_SALARY;
SET DEPT_BONUS_CNT = BONUS_CNT;
END P1;

```

This CREATE PROCEDURE statement:

- Names the procedure RETURN_DEPT_SALARY.
- Defines parameter DEPT_NUMBER which is an input parameter and is a character data type of length 3, parameter DEPT_SALARY which is an output parameter and is a decimal data type, and parameter DEPT_BONUS_CNT which is an output parameter and is an integer data type.
- Indicates the procedure is an SQL procedure that reads SQL data
- Defines the procedure body.
 - Declares SQL variables EMPLOYEE_SALARY and TOTAL_SALARY as decimal fields.
 - Declares SQL variables BONUS_CNT and END_TABLE which are integers and are initialized to 0.
 - Declares cursor C1 that selects the columns from the employee table.
 - Declares a continue handler for NOT FOUND, which, when invoked sets variable END_TABLE to 1. This handler is invoked when the FETCH has no more rows to return. When the handler is invoked, SQLCODE and SQLSTATE are reinitialized to 0.
 - Declares an exit handler for SQLEXCEPTION. If invoked, DEPT_SALARY is set to NULL and the processing of the compound statement is terminated. This handler is invoked if any errors occur, ie, the SQLSTATE class is not '00', '01' or '02'. Since indicators are always passed to SQL procedures, the indicator value for DEPT_SALARY is -1 when the procedure returns. If this handler is invoked, SQLCODE and SQLSTATE are reinitialized to 0.
 If the handler for SQLEXCEPTION is not specified and an error occurs that is not handled in another handler, execution of the compound statement is terminated and the error is returned in the SQLCA. Similar to indicators, the SQLCA is always returned from SQL procedures.
 - Includes an OPEN, FETCH, and CLOSE of cursor C1. If a CLOSE of the cursor is not specified, the cursor is closed at the end of the compound statement since SET RESULT SETS is not specified in the CREATE PROCEDURE statement.
 - Includes a WHILE statement which loops until the last record is fetched. For each row retrieved, the TOTAL_SALARY is incremented and, if the employee's bonus is more than 0, the BONUS_CNT is incremented.
 - Returns DEPT_SALARY and DEPT_BONUS_CNT as output parameters.

Compound statements can be made atomic so if an error occurs that is not expected, the statements within the atomic statement are rolled back. The atomic compound statements are implemented using SAVEPOINTS. If the compound statement is successful, the transaction is committed. For more information about using SAVEPOINTS, see "Savepoints" on page 314.

The following example takes as input the department number. It ensures the EMPLOYEE_BONUS table exists, and inserts the name of all employees in the department who get a bonus. The procedure returns the total count of all employees who get a bonus.

```

CREATE PROCEDURE CREATE_BONUS_TABLE
  (IN DEPT_NUMBER CHAR(3),
   INOUT CNT INT)
LANGUAGE SQL MODIFIES SQL DATA
  CS1: BEGIN ATOMIC
    DECLARE NAME VARCHAR(30) DEFAULT NULL;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '42710'
      SELECT COUNT(*) INTO CNT
      FROM DATALIB.EMPLOYEE_BONUS;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
      SET CNT = CNT - 1;
    DECLARE UNDO HANDLER FOR SQLEXCEPTION
      SET CNT = NULL;
    IF DEPT_NUMBER IS NOT NULL THEN
      CREATE TABLE DATALIB.EMPLOYEE_BONUS
        (FULLNAME VARCHAR(30),
         BONUS DECIMAL(10,2),
         PRIMARY KEY (FULLNAME));
    FOR_1:FOR V1 AS C1 CURSOR FOR
      SELECT FIRSTNAME, MIDINIT, LASTNAME, BONUS
      FROM CORPDATA.EMPLOYEE
      WHERE WORKDEPT = CREATE_BONUS_TABLE.DEPT_NUMBER
    DO
      IF BONUS > 0 THEN
        SET NAME = FIRSTNAME CONCAT ' ' CONCAT
          MIDINIT CONCAT ' ' CONCAT LASTNAME;
        INSERT INTO DATALIB.EMPLOYEE_BONUS
          VALUES(CS1.NAME, FOR_1.BONUS);
        SET CNT = CNT + 1;
      END IF;
    END FOR FOR_1;
  END IF;
END CS1

```

This CREATE PROCEDURE statement:

- Names the procedure CREATE_BONUS_TABLE.
- Defines parameter DEPT_NUMBER which is an input parameter and is a character data type of length 3 and parameter CNT which is an input/output parameter and is an integer data type.
- Indicates the procedure is an SQL procedure that modifies SQL data
- Defines the procedure body.
 - Declares SQL variable NAME as varying character.
 - Declares a continue handler for SQLSTATE 42710, table already exists. If the EMPLOYEE_BONUS table already exists, the handler is invoked and retrieves the number of records in the table. The SQLCODE and SQLSTATE are reset to 0 and processing continues with the FOR statement.
 - Declares a continue handler for SQLSTATE 23505, duplicate key. If the procedure attempts to insert a name that already exists in the table, the handler is invoked and decrements CNT. Processing continues on the SET statement following the INSERT statement.
 - Declares an UNDO handler for SQLEXCEPTION. If invoked, the previous statements are rolled back, CNT is set to 0, and processing continues after the compound statement. In this case, since there is no statement following the compound statement, the procedure returns.

- Uses the FOR statement to declare cursor C1 to read the records from the EMPLOYEE table. Within the FOR statement, the column names from the select list are used as SQL variables that contain the data from the row fetched. For each row, data from columns FIRSTNAME, MIDINIT, and LASTNAME are concatenated together with a blank in between and the result is put in SQL variable NAME. SQL variables NAME and BONUS are inserted into the EMPLOYEE_BONUS table. Because the data type of the select list items must be known when the procedure is created, the table specified in the FOR statement must exist when the procedure is created.

An SQL variable name can be qualified with the label name of the FOR statement or compound statement in which it is defined. In the example, FOR_1.BONUS refers to the SQL variable that contains the value of column BONUS for each row selected. CS1.NAME is the variable NAME defined in the compound statement with the beginning label CS1. Parameter names can also be qualified with the procedure name. CREATE_BONUS_TABLE.DEPT_NUMBER is the DEPT_NUMBER parameter for the procedure CREATE_BONUS_TABLE. If unqualified SQL variable names are used in SQL statements where column names are also allowed, and the variable name is the same as a column name, the name will be used to refer to the column.

You can also use dynamic SQL in an SQL procedure. The following example creates a table that contains all employees in a specific department. The department number is passed as input to the procedure and is concatenated to the table name.

```
CREATE PROCEDURE CREATE_DEPT_TABLE (IN P_DEPT CHAR(3))
LANGUAGE SQL
BEGIN
  DECLARE STMT CHAR(1000);
  DECLARE MESSAGE CHAR(20);
  DECLARE TABLE_NAME CHAR(30);
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SET MESSAGE = 'ok';
  SET TABLE_NAME = 'CORPDATA.DEPT_' CONCAT P_DEPT CONCAT '_T';
  SET STMT = 'DROP TABLE ' CONCAT TABLE_NAME;
  PREPARE S1 FROM STMT;
  EXECUTE S1;
  SET STMT = 'CREATE TABLE ' CONCAT TABLE_NAME CONCAT
    '( EMPNO CHAR(6) NOT NULL,
      FIRSTNAME VARCHAR(12) NOT NULL,
      MIDINIT CHAR(1) NOT NULL,
      LASTNAME CHAR(15) NOT NULL,
      SALARY DECIMAL(9,2))';
  PREPARE S2 FROM STMT;
  EXECUTE S2;
  SET STMT = 'INSERT INTO ' CONCAT TABLE_NAME CONCAT
    'SELECT EMPNO, FIRSTNAME, MIDINIT, LASTNAME, SALARY
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = ?';
  PREPARE S3 FROM STMT;
  EXECUTE S3 USING P_DEPT;
END;
```

This CREATE PROCEDURE statement:

- Names the procedure CREATE_DEPT_TABLE
- Defines parameter P_DEPT which is an input parameter and is a character data type of length 3.
- Indicates the procedure is an SQL procedure.
- Defines the procedure body.

- Declares SQL variable STMT and an SQL variable TABLE_NAME as character.
- Declares a CONTINUE handler. The procedure attempts to DROP the table in case it already exists. If the table does not exist, the first EXECUTE would fail. With the handler, processing will continue.
- Sets variable TABLE_NAME to 'DEPT_' followed by the characters passed in parameter P_DEPT, followed by '_T'.
- Sets variable STMT to the DROP statement, and prepares and executes the statement.
- Sets variable STMT to the CREATE statement, and prepares and executes the statement.
- Sets variable STMT to the INSERT statement, and prepares and executes the statement. A parameter marker is specified in the where clause. When the statement is executed, the variable P_DEPT is passed on the USING clause.

If the procedure is called passing value 'D21' for the department, table DEPT_D21_T is created and the table is initialized with all the employees that are in department 'D21'.

Debugging a stored procedure

By specifying SET OPTION DBGVIEW = *SOURCE in your Create SQL Procedure, Create SQL Function, or Create Trigger statement, you can debug the generated program or module at the SQL statement level. You can also specify DBGVIEW(*SOURCE) as a parameter on a RUNSQLSTM command and it will apply to all routines within the RUNSQLSTM.

The source view will be created by the system from your original routine body into QTEMP/QSQDSRC. The source view is not saved with the program or service program. It will be broken into lines that correspond to places you can stop in debug. The text, including parameter and variable names, will be folded to uppercase.

All variables and parameters are generated as part of a structure. The structure name must be used when evaluating a variable in debug. Variables are qualified by the current label name. Parameters are qualified by the procedure or function name. Transition variables in a trigger are qualified by the appropriate correlation name. It is highly recommended that you specify a label name for each compound statement or FOR statement. If you don't specify one, the system will generate one for you. This will make it nearly impossible to evaluate variables. Remember that all variables and parameters must be evaluated as an uppercase name. You can also eval the name of the structure. This will show you all the variables within the structure. If a variable or parameter is nullable, the indicator for that variable or parameter immediately follows it in the structure.

Because SQL routines are generated in C, there are some restrictions in C that also affect SQL source debug. Delimited names that are specified in the SQL routine body cannot be specified in C. Names are generated for these names, which again makes it difficult to debug or eval. In order to eval the contents of any character variable, specify an * prior to the name of the variable.

Since the system generates indicators for most variable and parameter names, there is no way to check directly to see if a variable has the SQL null value. Evaluating a variable will always show a value, even if the indicator is set to indicate the null value.

In order to determine if a handler is getting called, set a breakpoint on the first statement within the handler. Variables that are declared in a compound statement or FOR statement within the handler can be evaluated.

Invoking a stored procedure

The SQL CALL statement invokes a stored procedure. On the CALL statement, the name of the stored procedure and any arguments are specified. Arguments may be constants, special registers, or host variables. The external stored procedure specified in the CALL statement does not need to have a corresponding CREATE PROCEDURE statement. Programs created by SQL procedures can only be called by invoking the procedure name specified on the CREATE PROCEDURE statement.

Although procedures are system program objects, using the CALL CL command will not usually work to call a procedure. The CALL CL command does not use the procedure definition to map the input and output parameters, nor does it pass parameters to the program using the procedure's parameter style.

There are three types of CALL statements which need to be addressed since DB2 SQL for iSeries has different rules for each type. They are:

- Embedded or dynamic CALL statement where a procedure definition exists
- Embedded CALL statement where no procedure definition exists
- Dynamic CALL statement where no CREATE PROCEDURE exists

Note: Dynamic here refers to:

- A dynamically prepared and executed CALL statement
- A CALL statement issued in an interactive environment (for example, through STRSQL or Query Manager)
- A CALL statement executed in an EXECUTE IMMEDIATE statement.

Following is a discussion of each type.

- "Using CALL Statement where procedure definition exists"
- "Using Embedded CALL Statement where no procedure definition exists" on page 168
- "Using Embedded CALL statement with an SQLDA" on page 169
- "Using Dynamic CALL Statement where no CREATE PROCEDURE exists" on page 170

Using CALL Statement where procedure definition exists

This type of CALL statement reads all the information about the procedure and the argument attributes from the CREATE PROCEDURE catalog definition. The following PL/I example shows a CALL statement that corresponds to the CREATE PROCEDURE statement shown.

```
DCL HV1 CHAR(10);
DCL IND1 FIXED BIN(15);
:
EXEC SQL CREATE P1 PROCEDURE
      (INOUT PARM1 CHAR(10))
      EXTERNAL NAME MYLIB.PROC1
      LANGUAGE C
```

```

          GENERAL WITH NULLS;
      :
EXEC SQL CALL P1 (:HV1 :IND1);
      :

```

When this CALL statement is invoked, a call to program MYLIB/PROC1 is made and two arguments are passed. Since the language of the program is ILE C, the first argument is a C NUL-terminated string eleven characters long containing the contents of host variable HV1. Note that on a call to an ILE C procedure, DB2 SQL for iSeries adds one character to the parameter declaration if the parameter is declared to be a character, graphic, date, time, or timestamp variable. The second argument is the indicator array. In this case, it is one short integer since there is only one parameter in the CREATE PROCEDURE statement. This argument contains the contents of indicator variable IND1 on entry to the procedure.

Since the first parameter is declared as INOUT, SQL updates the host variable HV1 and the indicator variable IND1 with the values returned from MYLIB.PROC1 before returning to the user program.

Notes:

1. The procedure names specified on the CREATE PROCEDURE and CALL statements must match EXACTLY in order for the link between the two to be made during the SQL precompile of the program.
2. For an embedded CALL statement where both a CREATE PROCEDURE and a DECLARE PROCEDURE statement exist, the DECLARE PROCEDURE statement will be used.

Using Embedded CALL Statement where no procedure definition exists

A static CALL statement without a corresponding CREATE PROCEDURE statement is processed with the following rules:

- All host variable arguments are treated as INOUT type parameters.
- The CALL type is GENERAL (no indicator argument is passed).
- The program to call is determined based on the procedure name specified on the CALL, and, if necessary, the naming convention.
- The language of the program to call is determined based on information retrieved from the system about the program.

Example: Embedded CALL Statement Where No Procedure Definition Exists

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

The following is a PL/I example of an embedded CALL statement where no procedure definition exists:

```

DCL HV2 CHAR(10);
      :
EXEC SQL CALL P2 (:HV2);
      :

```

When the CALL statement is invoked, DB2 SQL for iSeries attempts to find the program based on standard SQL naming conventions. For the above example, assume that the naming option of *SYS (system naming) is used and that a DFTRDBCOL parameter was not specified on the CRTSQLPLI command. In this

case, the library list is searched for a program named P2. Since the call type is GENERAL, no additional argument is passed to the program for indicator variables.

Note: If an indicator variable is specified on the CALL statement and its value is less than zero when the CALL statement is executed, an error results because there is no way to pass the indicator to the procedure.

Assuming program P2 is found in the library list, the contents of host variable HV2 are passed in to the program on the CALL and the argument returned from P2 is mapped back to the host variable after P2 has completed execution.

Using Embedded CALL statement with an SQLDA

In either type of embedded CALL (where a procedure definition may or may not exist), an SQLDA may be passed rather than a parameter list, as illustrated in the following C example. Assume that the stored procedure is expecting 2 parameters, the first of type SHORT INT and the second of type CHAR with a length of 4.

```
#define SQLDA_HV_ENTRIES 2
#define SHORTINT 500
#define NUL_TERM_CHAR 460

exec sql include sqlca;
exec sql include sqlda;
...
typedef struct sqlda Sqlda;
typedef struct sqldap* Sqldap;
...
main()
{
    Sqldap dap;
    short col1;
    char col2[4];
    int bc;
    dap = (Sqldap) malloc(bc=SQLDASIZE(SQLDA_HV_ENTRIES));
        /* SQLDASIZE is a macro defined in the sqlda include */
    col1 = 431;
    strcpy(col2,"abc");
    strncpy(dap->sqldaid,"SQLDA ",8);
    dap->sqldabc = bc; /* bc set in the malloc statement above */
    dap->sqln = SQLDA_HV_ENTRIES;
    dap->sqld = SQLDA_HV_ENTRIES;
    dap->sqlvar[0].sqltype = SHORTINT;
    dap->sqlvar[0].sqllen = 2;
    dap->sqlvar[0].sqldata = (char*) &col1;
    dap->sqlvar[0].sqlname.length = 0;
    dap->sqlvar[1].sqltype = NUL_TERM_CHAR;
    dap->sqlvar[1].sqllen = 4;
    dap->sqlvar[1].sqldata = col2;
    ...
    EXEC SQL CALL P1 USING DESCRIPTOR :*dap;
    ...
}
```

The name of the called procedure may also be stored in a host variable and the host variable used in the CALL statement, instead of the hard-coded procedure name. For example:

```
...
main()
{
    char proc_name[15];
    ...
    strcpy (proc_name, "MYLIB.P3");
}
```

```

...
EXEC SQL CALL :proc_name ...;
...
}

```

In the above example, if MYLIB.P3 is expecting parameters, then either a parameter list or an SQLDA passed with the USING DESCRIPTOR clause may be used, as shown in the previous example.

When a host variable containing the procedure name is used in the CALL statement and a CREATE PROCEDURE catalog definition exists, it will be used. The procedure name cannot be specified as a parameter marker.

More examples for calling stored procedures may be found later in this chapter.

Using Dynamic CALL Statement where no CREATE PROCEDURE exists

The following rules pertain to the processing of a dynamic CALL statement when there is no CREATE PROCEDURE definition:

- All arguments are treated as IN type parameters.
- The CALL type is GENERAL (no indicator argument is passed).
- The program to call is determined based on the procedure name specified on the CALL and the naming convention.
- The language of the program to call is determined based on information retrieved from the system about the program.

Example: Dynamic CALL statement where no CREATE PROCEDURE exists

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

The following is a C example of a dynamic CALL statement:

```

char hv3[10],string[100];
:
strcpy(string,"CALL MYLIB.P3 ('P3 TEST')");
EXEC SQL EXECUTE IMMEDIATE :string;
:

```

This example shows a dynamic CALL statement executed through an EXECUTE IMMEDIATE statement. The call is made to program MYLIB.P3 with one parameter passed as a character variable containing 'P3 TEST'.

When executing a CALL statement and passing a constant, as in the previous example, the length of the expected argument in the program must be kept in mind. If program MYLIB.P3 expected an argument of only 5 characters, the last 2 characters of the constant specified in the example would be lost to the program.

Note: For this reason, it is always safer to use host variables on the CALL statement so that the attributes of the procedure can be matched exactly and so that characters are not lost. For dynamic SQL, host variables can be specified for CALL statement arguments if the PREPARE and EXECUTE statements are used to process it.

For numeric constants passed on a CALL statement, the following rules apply:

- All integer constants are passed as fullword binary integers.
- All decimal constants are passed as packed decimal values. Precision and scale are determined based on the constant value. For instance, a value of 123.45 is passed as a packed decimal(5,2). Likewise, a value of 001.01 is also passed with a precision and scale of 5 and 2, respectively.
- All floating point constants are passed as double-precision floating point.

Special registers specified on a dynamic CALL statement are passed as follows:

CURRENT DATE

Passed as a 10-byte character string in ISO format.

CURRENT TIME

Passed as an 8-byte character string in ISO format.

CURRENT TIMESTAMP

Passed as a 26-byte character string in IBM SQL format.

CURRENT TIMEZONE

Passed as a packed decimal number with a precision of 6 and a scale of 0.

CURRENT SCHEMA

Passed as an 128-byte varying length character string

CURRENT SERVER

Passed as an 18-byte varying length character string.

USER

Passed as an 18-byte varying length character string.

CURRENT PATH

Passed as a 3483-byte varying length character string.

Parameter passing conventions for stored procedures and UDFs

The CALL statement can pass arguments to programs written in all supported host languages and REXX procedures. Each language supports different data types that are tailored to it. The SQL data type is contained in the leftmost column of the following table. Other columns in that row contain an indication of whether that data type is supported as a parameter type for a particular language. If the column contains a dash (-), the data type is not supported as a parameter type for that language. A host variable declaration indicates that DB2 SQL for iSeries supports this data type as a parameter in this language. The declaration indicates how host variables must be declared to be received and set properly by the procedure. When calling an SQL procedure, all SQL data types are supported so no column is provided in the table.

See the SQL Programming for Host Languages book and the Java SQL routines section of the IBM Developer's Kit for Java topic for more details.

Table 25. Data Types of Parameters

SQL Data Type	C and C++	CL	COBOL for iSeries and ILE COBOL for iSeries
SMALLINT	short	-	PIC S9(4) BINARY
INTEGER	long	-	PIC S9(9) BINARY
BIGINT	long long	-	PIC S9(18) BINARY Note: Only supported for ILE COBOL for iSeries.

Table 25. Data Types of Parameters (continued)

SQL Data Type	C and C++	CL	COBOL for iSeries and ILE COBOL for iSeries
DECIMAL(p,s)	decimal(p,s)	TYPE(*DEC) LEN(p s)	PIC S9(p-s)V9(s) PACKED-DECIMAL Note: Precision must not be greater than 18.
NUMERIC(p,s)	-	-	PIC S9(p-s)V9(s) DISPLAY SIGN LEADING SEPARATE Note: Precision must not be greater than 18.
REAL or FLOAT(p)	float	-	COMP-1 Note: Only supported for ILE COBOL for iSeries.
DOUBLE PRECISION or FLOAT or FLOAT(p)	double	-	COMP-2 Note: Only supported for ILE COBOL for iSeries.
CHARACTER(n)	char ... [n+1]	TYPE(*CHAR) LEN(n)	PIC X(n)
VARCHAR(n)	char ... [n+1]	-	Varying-Length Character String (see COBOL chapter in SQL Programming with Host Languages).
VARCHAR(n) FOR BIT DATA	VARCHAR structured form (see C chapter in SQL Programming for Host Languages book.)	-	Varying-Length Character String (see COBOL chapter in SQL Programming with Host Languages).
GRAPHIC(n)	wchar_t ... [n+1]	-	PIC G(n) DISPLAY-1 or PIC N(n) Note: Only supported for ILE COBOL for iSeries.
VARGRAPHIC(n)	VARGRAPHIC structured form (see C chapter)	-	Varying-Length Graphic String (see COBOL chapter in SQL Programming with Host Languages). Note: Only supported for ILE COBOL for iSeries.
DATE	char ... [11]	TYPE(*CHAR) LEN(10)	PIC X(10) For ILE COBOL for iSeries only, FORMAT DATE.
TIME	char ... [9]	TYPE(*CHAR) LEN(8)	PIC X(8) For ILE COBOL for iSeries only, FORMAT TIME.
TIMESTAMP	char ... [27]	TYPE(*CHAR) LEN(26)	PIC X(26) For ILE COBOL for iSeries only, FORMAT TIMESTAMP.
CLOB	CLOB structured form (see C chapter in SQL Programming with Host Languages)	-	CLOB structured form (see COBOL chapter in SQL Programming with Host Languages). Note: only supported for ILE COBOL for iSeries.

Table 25. Data Types of Parameters (continued)

SQL Data Type	C and C++	CL	COBOL for iSeries and ILE COBOL for iSeries
BLOB	BLOB structured form (see C chapter in SQL Programming with Host Languages)	-	BLOB structured form (see COBOL chapter in SQL Programming with Host Languages). Note: only supported for ILE COBOL for iSeries.
DBCLOB	DBCLOB structured form (see C chapter in SQL Programming with Host Languages)	-	DBCLOB structured form (see COBOL chapter in SQL Programming with Host Languages). Note: only supported for ILE COBOL for iSeries.
ROWID	ROWID structured form (see C chapter in SQL Programming with Host Languages)	-	ROWID structured form (see COBOL chapter in SQL Programming with Host Languages).
DataLink	-	-	-
Indicator Variable	short	-	PIC S9(4) BINARY

Table 26. Data Types of Parameters

SQL Data Type	FORTRAN	Java Parameter Style JAVA	Java Parameter Style DB2GENERAL	PL/I
SMALLINT	INTEGER*2	short	short	FIXED BIN(15)
INTEGER	INTEGER*4	int	int	FIXED BIN(31)
BIGINT	-	long	long	-
DECIMAL(p,s)	-	BigDecimal	BigDecimal	FIXED DEC(p,s)
NUMERIC(p,s)	-	BigDecimal	BigDecimal	-
REAL or FLOAT(p)	REAL*4	float	float	FLOAT BIN(p)
DOUBLE PRECISION or FLOAT or FLOAT(p)	REAL*8	double	double	FLOAT BIN(p)
CHARACTER(n)	CHARACTER*n	String	String	CHAR(n)
VARCHAR(n)	-	String	String	CHAR(n) VAR
VARCHAR(n) FOR BIT DATA	-	-	com.ibm.db2.app.Blob	CHAR(n) VAR
GRAPHIC(n)	-	String	String	-
VARGRAPHIC(n)	-	String	String	-
DATE	CHARACTER*10	Date	String	CHAR(10)
TIME	CHARACTER*8	Time	String	CHAR(8)
TIMESTAMP	CHARACTER*26	Timestamp	String	CHAR(26)
CLOB	-	-	com.ibm.db2.app.Clob	CLOB structured form (see PL/I chapter in SQL Programming with Host Languages)

Table 26. Data Types of Parameters (continued)

SQL Data Type	FORTTRAN	Java Parameter Style JAVA	Java Parameter Style DB2GENERAL	PL/I
BLOB	-	-	com.ibm.db2.app.Blob	BLOB structured form (see PL/I chapter in SQL Programming with Host Languages)
DBCLOB	-	-	com.ibm.db2.app.Clob	DBCLOB structured form (see PL/I chapter in SQL Programming with Host Languages)
ROWID	-	-	-	ROWID structured form (see PL/I chapter in SQL Programming with Host Languages)
DataLink	-	-	-	-
Indicator Variable	INTEGER*2	-	-	FIXED BIN(15)

Table 27. Data Types of Parameters

SQL Data Type	REXX	RPG	ILE RPG
SMALLINT	-	Data structure that contains a single sub-field. <i>B</i> in position 43, length must be 2, and 0 in position 52 of the sub-field specification.	Data specification. <i>B</i> in position 40, length must be <= 4, and 00 in positions 41-42 of the sub-field specification. or Data specification. <i>I</i> in position 40, length must be 5, and 00 in positions 41-42 of the sub-field specification.
INTEGER	numeric string with no decimal (and an optional leading sign)	Data structure that contains a single sub-field. <i>B</i> in position 43, length must be 4, and 0 in position 52 of the sub-field specification.	Data specification. <i>B</i> in position 40, length must be <=09 and >=05, and 00 in positions 41-42 of the sub-field specification. or Data specification. <i>I</i> in position 40, length must be 10, and 00 in positions 41-42 of the sub-field specification.
BIGINT	-	-	Data specification. <i>I</i> in position 40, length must be 20, and 00 in positions 41-42 of the sub-field specification.
DECIMAL(p,s)	numeric string with a decimal (and an optional leading sign)	Data structure that contains a single sub-field. <i>P</i> in position 43 and 0 through 9 in position 52 of the sub-field specification. or A numeric input field or calculation result field.	Data specification. <i>P</i> in position 40 and 00 through 31 in positions 41-42 of the sub-field specification.

Table 27. Data Types of Parameters (continued)

SQL Data Type	REXX	RPG	ILE RPG
NUMERIC(p,s)	-	Data structure that contains a single sub-field. <i>Blank</i> in position 43 and 0 through 9 in position 52 of the sub-field specification.	Data specification. <i>S</i> in position 40, or <i>Blank</i> in position 40 and 00 through 31 in position 41-42 of the sub-field specification.
REAL or FLOAT(p)	string with digits, then an E, (then an optional sign), then digits	-	Data specification. <i>F</i> in position 40, length must be 4.
DOUBLE PRECISION or FLOAT or FLOAT(p)	string with digits, then an E, (then an optional sign), then digits	-	Data specification. <i>F</i> in position 40, length must be 8.
CHARACTER(n)	string with n characters within two apostrophes	Data structure field without sub-fields or data structure that contains a single sub-field. <i>Blank</i> in position 43 and 52 of the sub-field specification. or A character input field or calculation result field.	Data specification. <i>A</i> in position 40, or <i>Blank</i> in position 40 and 41-42 of the sub-field specification.
VARCHAR(n)	string with n characters within two apostrophes	-	Data specification. <i>A</i> in position 40, or <i>Blank</i> in position 40 and 41-42 of the sub-field specification and the keyword VARYING in positions 44-80.
VARCHAR(n) FOR BIT DATA	string with n characters within two apostrophes	-	Data specification. <i>A</i> in position 40, or <i>Blank</i> in position 40 and 41-42 of the sub-field specification and the keyword VARYING in positions 44-80.
GRAPHIC(n)	string starting with G', then n double byte characters, then '	-	Data specification. <i>G</i> in position 40 of the sub-field specification.
VARGRAPHIC(n)	string starting with G', then n double byte characters, then '	-	Data specification. <i>G</i> in position 40 of the sub-field specification and the keyword VARYING in positions 44-80.
DATE	string with 10 characters within two apostrophes	Data structure field without sub-fields or data structure that contains a single sub-field. <i>Blank</i> in position 43 and 52 of the sub-field specification. Length is 10. or A character input field or calculation result field.	Data specification. <i>D</i> in position 40 of the sub-field specification. DATFMT(*ISO) in position 44-80.
TIME	string with 8 characters within two apostrophes	Data structure field without sub-fields or data structure that contains a single sub-field. <i>Blank</i> in position 43 and 52 of the sub-field specification. Length is 8. or A character input field or calculation result field.	Data specification. <i>T</i> in position 40 of the sub-field specification. TIMFMT(*ISO) in position 44-80.

Table 27. Data Types of Parameters (continued)

SQL Data Type	REXX	RPG	ILE RPG
TIMESTAMP	string with 26 characters within two apostrophes	Data structure field without sub-fields or data structure that contains a single sub-field. <i>Blank</i> in position 43 and 52 of the sub-field specification. Length is 26. or A character input field or calculation result field.	Data specification. <i>Z</i> in position 40 of the sub-field specification.
CLOB	-	-	CLOB structured form (see RPG chapter in SQL Programming with Host Languages)
BLOB	-	-	BLOB structured form (see RPG chapter in SQL Programming with Host Languages)
DBCLOB	-	-	DBCLOB structured form (see RPG chapter in SQL Programming with Host Languages)
ROWID	-	-	ROWID structured form (see RPG chapter in SQL Programming with Host Languages)
DataLink	-	-	-
Indicator Variable	numeric string with no decimal (and an optional leading sign).	Data structure that contains a single sub-field. <i>B</i> in position 43, length must be 2, and 0 in position 52 of the sub-field specification.	Data specification. <i>B</i> in position 40, length must be <=4, and 00 in positions 41-42 of the sub-field specification.

Indicator variables and stored procedures

Indicator variables can be used with the CALL statement, provided host variables are used for the parameters, to pass additional information to and from the procedure. Indicator variables are the SQL standard means of denoting that the associated host variable should be interpreted as containing the null value, and this is their primary use.

To indicate that an associated host variable contains the null value, the indicator variable, which is a two-byte integer, is set to a negative value. A CALL statement with indicator variables is processed as follows:

- If the indicator variable is negative, this denotes the null value. A default value is passed for the associated host variable on the CALL and the indicator variable is passed unchanged.
- If the indicator variable is not negative, this denotes that the host variable contains a non-null value. In this case, the host variable and the indicator variable are passed unchanged.

These rules of processing are the same for input parameters to the procedure as well as output parameters returned from the procedure. When indicator variables are used with stored procedures, the correct method of coding their handling is to check the value of the indicator variable first before using the associated host variable.

The following example illustrates the handling of indicator variables in CALL statements. Notice that the logic checks the value of the indicator variable before using the associated variable. Also note the method that the indicator variables are passed into procedure PROC1 (as a third argument consisting of an array of two-byte values).

Assume a procedure was defined as follows:

```
CREATE PROCEDURE PROC1
  (INOUT DECIMALOUT DECIMAL(7,2), INOUT DECOUT2 DECIMAL(7,2))
  EXTERNAL NAME LIB1.PROC1 LANGUAGE RPGLE
  GENERAL WITH NULLS)
```

```
+++++
Program CRPG
+++++
D INOUT1          S          7P 2
D INOUT1IND       S          4B 0
D INOUT2          S          7P 2
D INOUT2IND       S          4B 0
C
C          EVAL      INOUT1 = 1
C          EVAL      INOUT1IND = 0
C          EVAL      INOUT2 = 1
C          EVAL      INOUT2IND = -2
C/EXEC SQL CALL PROC1 (:INOUT1 :INOUT1IND , :INOUT2
C+                :INOUT2IND)
C/END-EXEC
C          EVAL      INOUT1 = 1
C          EVAL      INOUT1IND = 0
C          EVAL      INOUT2 = 1
C          EVAL      INOUT2IND = -2
C/EXEC SQL CALL PROC1 (:INOUT1 :INOUT1IND , :INOUT2
C+                :INOUT2IND)
C/END-EXEC
C  INOUT1IND      IFLT      0
C*
C*                :
C*                HANDLE NULL INDICATOR
C*                :
C                ELSE
C*                :
C*                INOUT1 CONTAINS VALID DATA
C*                :
C                ENDIF
C*                :
C*                HANDLE ALL OTHER PARAMETERS
C*                IN A SIMILAR FASHION
C*                :
C                RETURN
+++++
End of PROGRAM CRPG
+++++
```

Figure 3. Handling of Indicator Variables in CALL Statements (Part 1 of 2)

```

+++++
Program PROC1
+++++
D INOUTP          S          7P 2
D INOUTP2         S          7P 2
D NULLARRAY       S          4B 0 DIM(2)
C   *ENTRY        PLIST
C                 PARM          INOUTP
C                 PARM          INOUTP2
C                 PARM          NULLARRAY
C   NULLARRAY(1)  IFLT      0
C*                :
C*                INOUTP DOES NOT CONTAIN MEANINGFUL DATA
C*
C                 ELSE
C*                :
C*                INOUTP CONTAINS MEANINGFUL DATA
C*                :
C                 ENDFIF
C*                PROCESS ALL REMAINING VARIABLES
C*
C*                BEFORE RETURNING, SET OUTPUT VALUE FOR FIRST
C*                PARAMETER AND SET THE INDICATOR TO A NON-NEGATIV
C*                VALUE SO THAT THE DATA IS RETURNED TO THE CALLING
C*                PROGRAM
C*
C                 EVAL      INOUTP2 = 20.5
C                 EVAL      NULLARRAY(2) = 0
C*
C*                INDICATE THAT THE SECOND PARAMETER IS TO CONTAIN
C*                THE NULL VALUE UPON RETURN. THERE IS NO POINT
C*                IN SETTING THE VALUE IN INOUTP SINCE IT WON'T BE
C*                PASSED BACK TO THE CALLER.
C                 EVAL      NULLARRAY(1) = -5
C                 RETURN
+++++
End of PROGRAM PROC1
+++++

```

Figure 3. Handling of Indicator Variables in CALL Statements (Part 2 of 2)

Returning a completion status to the calling program

For SQL procedures, any errors that are not handled in the procedure are returned to the caller in the SQLCA. The SIGNAL and RESIGNAL control statements can be used to send error information as well. See the SQL Procedures, Functions, and Triggers topic in the SQL Reference for more information.

For external procedures, there are two ways to return status information. One method of returning a status to the SQL program issuing the CALL statement is to code an extra INOUT type parameter and set it prior to returning from the procedure. When the procedure being called is an existing program, this is not always possible.

Another method of returning a status to the SQL program issuing the CALL statement is to send an escape message to the calling program (operating system program QSQCALL) which invokes the procedure. The calling program that invokes the procedure is QSQCALL. Each language has methods for signalling conditions and sending messages. Refer to the respective language reference to

determine the proper way to signal a message. When the message is signalled, QSQCALL turns the error into SQLCODE/SQLSTATE -443/38501.

Examples of CALL statements

These examples show how the arguments of the CALL statement are passed to the procedure for several languages. They also show how to receive the arguments into local variables in the procedure.

The first example shows the calling ILE C program that uses the CREATE PROCEDURE definitions to call the P1 and P2 procedures. Procedure P1 is written in C and has 10 parameters. Procedure P2 is written in PL/I and also has 10 parameters.

Assume two procedures are defined as follows:

```
EXEC SQL CREATE PROCEDURE P1 (INOUT PARM1 CHAR(10),
                              INOUT PARM2 INTEGER,
                              INOUT PARM3 SMALLINT,
                              INOUT PARM4 FLOAT(22),
                              INOUT PARM5 FLOAT(53),
                              INOUT PARM6 DECIMAL(10,5),
                              INOUT PARM7 VARCHAR(10),
                              INOUT PARM8 DATE,
                              INOUT PARM9 TIME,
                              INOUT PARM10 TIMESTAMP)
      EXTERNAL NAME TEST12.CALLPROC2
      LANGUAGE C GENERAL WITH NULLS

EXEC SQL CREATE PROCEDURE P2 (INOUT PARM1 CHAR(10),
                              INOUT PARM2 INTEGER,
                              INOUT PARM3 SMALLINT,
                              INOUT PARM4 FLOAT(22),
                              INOUT PARM5 FLOAT(53),
                              INOUT PARM6 DECIMAL(10,5),
                              INOUT PARM7 VARCHAR(10),
                              INOUT PARM8 DATE,
                              INOUT PARM9 TIME,
                              INOUT PARM10 TIMESTAMP)
      EXTERNAL NAME TEST12.CALLPROC
      LANGUAGE PLI GENERAL WITH NULLS
```

Example 1: ILE C and PL/I procedures called from ILE C applications

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

```

/*****
/***** START OF SQL C Application *****/

#include <stdio.h>
#include <string.h>
#include <decimal.h>
main()
{
EXEC SQL INCLUDE SQLCA;
char PARM1[10];
signed long int PARM2;
signed short int PARM3;
float PARM4;
double PARM5;
decimal(10,5) PARM6;
struct { signed short int parm7l;
        char parm7c[10];
        } PARM7;
char PARM8[10];      /* FOR DATE */
char PARM9[8];      /* FOR TIME */
char PARM10[26];    /* FOR TIMESTAMP */

```

Figure 4. Sample of CREATE PROCEDURE and CALL (Part 1 of 2)

```

/*****
/* Initialize variables for the call to the procedures */
/*****
strcpy(PARM1,"PARM1");
PARM2 = 7000;
PARM3 = -1;
PARM4 = 1.2;
PARM5 = 1.0;
PARM6 = 10.555;
PARM7.parm7l = 5;
strcpy(PARM7.parm7c,"PARM7");
strncpy(PARM8,"1994-12-31",10);          /* FOR DATE      */
strncpy(PARM9,"12.00.00",8);           /* FOR TIME      */
strncpy(PARM10,"1994-12-31-12.00.00.000000",26);
                                          /* FOR TIMESTAMP */
/*****
/* Call the C procedure                    */
/*                                        */
/*                                        */
/*****
EXEC SQL CALL P1 (:PARM1, :PARM2, :PARM3,
                 :PARM4, :PARM5, :PARM6,
                 :PARM7, :PARM8, :PARM9,
                 :PARM10 );
if (strncmp(SQLSTATE,"00000",5))
{
/* Handle error or warning returned on CALL statement */
}

/* Process return values from the CALL.          */
:

/*****
/* Call the PLI procedure                    */
/*                                        */
/*                                        */
/*****
/* Reset the host variables prior to making the CALL */
/*                                        */
:
EXEC SQL CALL P2 (:PARM1, :PARM2, :PARM3,
                 :PARM4, :PARM5, :PARM6,
                 :PARM7, :PARM8, :PARM9,
                 :PARM10 );
if (strncmp(SQLSTATE,"00000",5))
{
/* Handle error or warning returned on CALL statement */
}
/* Process return values from the CALL.          */
:
}

/***** END OF C APPLICATION *****/
/*****

```

Figure 4. Sample of CREATE PROCEDURE and CALL (Part 2 of 2)

```

/***** START OF C PROCEDURE P1 *****/
/*      PROGRAM TEST12/CALLPROC2      */
/*****

#include <stdio.h>
#include <string.h>
#include <decimal.h>
main(argc,argv)
  int argc;
  char *argv[];
  {
  char parm1[11];
  long int parm2;
  short int parm3,i,j,*ind,ind1,ind2,ind3,ind4,ind5,ind6,ind7,
      ind8,ind9,ind10;
  float parm4;
  double parm5;
  decimal(10,5) parm6;
  char parm7[11];
  char parm8[10];
  char parm9[8];
  char parm10[26];
  /* *****/
  /* Receive the parameters into the local variables - */
  /* Character, date, time, and timestamp are passed as */
  /* NUL terminated strings - cast the argument vector to */
  /* the proper data type for each variable. Note that */
  /* the argument vector could be used directly instead of */
  /* copying the parameters into local variables - the copy */
  /* is done here just to illustrate the method. */
  /* *****/

  /* Copy 10 byte character string into local variable */
  strcpy(parm1,argv[1]);

  /* Copy 4 byte integer into local variable */
  parm2 = *(int *) argv[2];

  /* Copy 2 byte integer into local variable */
  parm3 = *(short int *) argv[3];

  /* Copy floating point number into local variable */
  parm4 = *(float *) argv[4];

  /* Copy double precision number into local variable */
  parm5 = *(double *) argv[5];

  /* Copy decimal number into local variable */
  parm6 = *(decimal(10,5) *) argv[6];

```

Figure 5. Sample Procedure P1 (Part 1 of 2)

```

/*****
/* Copy NUL terminated string into local variable. */
/* Note that the parameter in the CREATE PROCEDURE was */
/* declared as varying length character. For C, varying */
/* length are passed as NUL terminated strings unless */
/* FOR BIT DATA is specified in the CREATE PROCEDURE */
*****/
strcpy(parm7,argv[7]);

/*****
/* Copy date into local variable. */
/* Note that date and time variables are always passed in */
/* ISO format so that the lengths of the strings are */
/* known. strcpy would work here just as well. */
*****/
strncpy(parm8,argv[8],10);

/* Copy time into local variable */
strncpy(parm9,argv[9],8);

/*****
/* Copy timestamp into local variable. */
/* IBM SQL timestamp format is always passed so the length*/
/* of the string is known. */
*****/
strncpy(parm10,argv[10],26);

/*****
/* The indicator array is passed as an array of short */
/* integers. There is one entry for each parameter passed */
/* on the CREATE PROCEDURE (10 for this example). */
/* Below is one way to set each indicator into separate */
/* variables. */
*****/
ind = (short int *) argv[11];
ind1 = *(ind++);
ind2 = *(ind++);
ind3 = *(ind++);
ind4 = *(ind++);
ind5 = *(ind++);
ind6 = *(ind++);
ind7 = *(ind++);
ind8 = *(ind++);
ind9 = *(ind++);
ind10 = *(ind++);
:
/* Perform any additional processing here */
:
return;
}
/***** END OF C PROCEDURE P1 *****/

```

Figure 5. Sample Procedure P1 (Part 2 of 2)

```

/***** START OF PL/I PROCEDURE P2 *****/
/***** PROGRAM TEST12/CALLPROC *****/
/*****/

CALLPROC :PROC( PARM1,PARM2,PARM3,PARM4,PARM5,PARM6,PARM7,
                PARM8,PARM9,PARM10,PARM11);
DCL SYSPRINT FILE STREAM OUTPUT EXTERNAL;
OPEN FILE(SYSPRINT);
DCL PARM1 CHAR(10);
DCL PARM2 FIXED BIN(31);
DCL PARM3 FIXED BIN(15);
DCL PARM4 BIN FLOAT(22);
DCL PARM5 BIN FLOAT(53);
DCL PARM6 FIXED DEC(10,5);
DCL PARM7 CHARACTER(10) VARYING;
DCL PARM8 CHAR(10);      /* FOR DATE */
DCL PARM9 CHAR(8);      /* FOR TIME */
DCL PARM10 CHAR(26);    /* FOR TIMESTAMP */
DCL PARM11(10) FIXED BIN(15); /* Indicators */

/* PERFORM LOGIC - Variables can be set to other values for */
/* return to the calling program.                               */

:

END CALLPROC;

```

Figure 6. Sample Procedure P2

The next example shows a REXX procedure called from an ILE C program.

Assume a procedure is defined as follows:

```

EXEC SQL CREATE PROCEDURE REXXPROC
      (IN PARM1 CHARACTER(20),
       IN PARM2 INTEGER,
       IN PARM3 DECIMAL(10,5),
       IN PARM4 DOUBLE PRECISION,
       IN PARM5 VARCHAR(10),
       IN PARM6 GRAPHIC(4),
       IN PARM7 VARGRAPHIC(10),
       IN PARM8 DATE,
       IN PARM9 TIME,
       IN PARM10 TIMESTAMP)
      EXTERNAL NAME 'TEST.CALLSRC(CALLREXX)'
      LANGUAGE REXX GENERAL WITH NULLS

```

Example 2. Sample REXX Procedure Called From C Application

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

```

/*****
/***** START OF SQL C Application *****/

#include <decimal.h>
#include <stdio.h>
#include <string.h>
#include <wchar.h>
/*-----*/
exec sql include sqlca;
exec sql include sqllda;
/* *****/
/* Declare host variable for the CALL statement */
/* *****/
char parm1[20];
signed long int parm2;
decimal(10,5) parm3;
double parm4;
struct { short dlen;
        char dat[10];
        } parm5;
wchar_t parm6[4] = { 0xC1C1, 0xC2C2, 0xC3C3, 0x0000 };
struct { short dlen;
        wchar_t dat[10];
        } parm7 = {0x0009, 0xE2E2,0xE3E3,0xE4E4, 0xE5E5, 0xE6E6,
                  0xE7E7, 0xE8E8, 0xE9E9, 0xC1C1, 0x0000 };

char parm8[10];
char parm9[8];
char parm10[26];
main()
{

```

Figure 7. Sample REXX Procedure Called From C Application (Part 1 of 4)

```

/* *****/
/* Call the procedure - on return from the CALL statement the */
/* SQLCODE should be 0. If the SQLCODE is non-zero,          */
/* the procedure detected an error.                          */
/* *****/
strcpy(parm1,"TestingREXX");
parm2 = 12345;
parm3 = 5.5;
parm4 = 3e3;
parm5.dlen = 5;
strcpy(parm5.dat,"parm6");
strcpy(parm8,"1994-01-01");
strcpy(parm9,"13.01.00");
strcpy(parm10,"1994-01-01-13.01.00.000000");

EXEC SQL CALL REXXPROC (:parm1, :parm2,
                       :parm3,:parm4,
                       :parm5, :parm6,
                       :parm7,
                       :parm8, :parm9,
                       :parm10);

if (strncpy(SQLSTATE,"00000",5))
{
  /* handle error or warning returned on CALL */
  :
}
:
}

/***** END OF SQL C APPLICATION *****/
/*****/

```

Figure 7. Sample REXX Procedure Called From C Application (Part 2 of 4)


```

/***** START OF REXX MEMBER TEST/CALLSRC CALLREXX *****/
/***** REXX source member TEST/CALLSRC CALLREXX */
/* Note the extra parameter being passed for the indicator*/
/* array. */
/* */
/* ACCEPT THE FOLLOWING INPUT VARIABLES SET TO THE */
/* SPECIFIED VALUES : */
/* AR1 CHAR(20) = 'TestingREXX' */
/* AR2 INTEGER = 12345 */
/* AR3 DECIMAL(10,5) = 5.5 */
/* AR4 DOUBLE PRECISION = 3e3 */
/* AR5 VARCHAR(10) = 'parm6' */
/* AR6 GRAPHIC = G'C1C1C2C2C3C3' */
/* AR7 VARGRAPHIC = */
/* G'E2E2E3E3E4E4E5E5E6E6E7E7E8E8E9E9EAEA' */
/* AR8 DATE = '1994-01-01' */
/* AR9 TIME = '13.01.00' */
/* AR10 TIMESTAMP = */
/* '1994-01-01-13.01.00.000000' */
/* AR11 INDICATOR ARRAY = +0+0+0+0+0+0+0+0+0+0 */

/*****
/* Parse the arguments into individual parameters */
/*****
parse arg ar1 ar2 ar3 ar4 ar5 ar6 ar7 ar8 ar9 ar10 ar11

/*****
/* Verify that the values are as expected */
/*****
if ar1<>'TestingREXX' then signal ar1tag
if ar2<>12345 then signal ar2tag
if ar3<>5.5 then signal ar3tag
if ar4<>3e3 then signal ar4tag
if ar5<>'parm6' then signal ar5tag
if ar6 <>'G'AABBCC' then signal ar6tag
if ar7 <>'G'SSTTUUVVWXXYYZZAA' then ,
signal ar7tag
if ar8 <> '1994-01-01' then signal ar8tag
if ar9 <> '13.01.00' then signal ar9tag
if ar10 <> '1994-01-01-13.01.00.000000' then signal ar10tag
if ar11 <> '+0+0+0+0+0+0+0+0+0+0' then signal ar11tag

```

Figure 7. Sample REXX Procedure Called From C Application (Part 3 of 4)

```

/*****
/* Perform other processing as necessary ..          */
/*****
:
/*****
/* Indicate the call was successful by exiting with a    */
/* return code of 0                                     */
/*****
exit(0)

ar1tag:
say "ar1 did not match" ar1
exit(1)
ar2tag:
say "ar2 did not match" ar2
exit(1)
:
:

/***** END OF REXX MEMBER *****/

```

Figure 7. Sample REXX Procedure Called From C Application (Part 4 of 4)

Chapter 12. Using the Object-Relational Capabilities

This chapter discusses the object-oriented capabilities of DB2.

- Why use the DB2 object extensions?
- DB2 approach to supporting objects
- Using Large Objects (LOBs)
- User-defined functions (UDF)
- User-defined distinct types (UDT)
- Synergy between UDTs, UDFs, and LOBs

One of the most important recent developments in modern programming language technology is *object-orientation*. Object orientation is the notion that entities in the application domain can be modeled as independent objects that are related to one another by means of classification. Object-orientation lets you capture the similarities and differences among objects in your application domain and group those objects together into related types. Objects of the same type behave in the same way because they share the same set of type-specific functions. These functions reflect the behavior of your objects in the application domain.

Why use the DB2 object extensions?

With the object extensions of DB2, you can incorporate object-oriented (OO) concepts and methodologies into your relational database. You accomplish this by extending it with richer sets of types and functions. With these extensions, you can store instances of object-oriented data types in columns of tables, and operate on them by means of functions in SQL statements. In addition, you can make the semantic behavior of stored objects an important resource that can be shared among all your applications by means of the database.

To incorporate object-orientation into your relational database systems, you can define new types and functions of your own. These new types and functions should reflect the semantics of the objects in your application domain. Some of the data objects you want to model may be large and complex (for example, text, voice, image, and financial data). Therefore, you may also need mechanisms for the storage and manipulation of large objects. User-defined Distinct types (UDTs), user-defined functions (UDFs), and Large Objects (LOBs) are the mechanisms that are provided by DB2. With DB2, you can now define new types and functions of your own to store and manipulate application objects within the database.

As described in subsequent sections, there is an important synergy among these object-oriented features. You can model a complex object in the application domain as a UDT. The UDT may in turn be internally represented as a LOB. In turn, the UDT's behavior may be implemented in terms of UDFs. This section shows you how to use LOBs along with the steps that are required to define UDTs and UDFs. You will also learn how UDTs, UDFs, and LOBs can better represent the objects in your application and thus work together.

Note: The use of the DB2 object-oriented mechanisms (UDTs, UDFs, and LOBs) is not restricted to the support of object-oriented applications. Just as the C++ programming language implements all sorts of non-object-oriented applications, the object-oriented mechanisms provided by DB2 can also support all kinds of non-object-oriented applications. UDTs, UDFs, and LOBs are general-purpose mechanisms that can be used to model any

database application. For this reason, these DB2 object extensions offer extensive support for both non-traditional, that is, object-oriented applications, in addition to improving support for traditional ones.

DB2 approach to supporting objects

The object extensions of DB2 enable you to realize the benefits of object technology while building on the strengths of relational technology. In a relational system, data types describe the data stored in columns of tables where the instances (or objects) of these data types are stored. Operations on these instances are supported by means of operators or functions that can be invoked anywhere that expressions are allowed.

The DB2 approach to support object extensions fits exactly into the relational paradigm. UDTs are data types that you define. UDT's, like built-in types, can be used to describe the data that is stored in columns of tables. UDFs are functions that you define. UDFs, like built-in functions or operators, support the manipulation of UDT instances. Thus, UDT instances are stored in columns of tables and manipulated by UDFs in SQL queries. UDTs can be internally represented in different ways. LOBs are just one example of this.

Using Large Objects (LOBs)

The VARCHAR and VARGRAPHIC data types have a limit of 32K bytes of storage. While this may be sufficient for small to medium size text data, applications often need to store large text documents. They may also need to store a wide variety of additional data types such as audio, video, drawings, mixed text and graphics, and images. DB2 provides three data types to store these data objects as strings of up to two (2) gigabytes (GB) in size. The three data types are: Binary Large Objects (BLOBs), single-byte Character Large Objects (CLOBs), and Double-Byte Character Large Objects (DBCLOBs).

Along with storing large objects (LOBs), you will also need a method to refer to, use, and modify each LOB in the database. Each DB2 table may have a large amount of associated LOB data. Although a single row containing one or more LOB values cannot exceed 3.5 gigabytes, a table may contain nearly 256 gigabytes of LOB data. The content of the LOB column of a particular row at any point in time has a *large object value*.

You can refer to and manipulate LOBs using host variables just as you would any other data type. However, host variables use the client memory buffer which may not be large enough to hold LOB values. Other means are necessary to manipulate these large values. *Locators* are useful to identify and manipulate a large object value at the database server and for extracting pieces of the LOB value. *File reference variables* are useful for physically moving a large object value (or a large part of it) to and from the client.

The subsections that follow discuss the topics that are introduced above in more detail:

- “Understanding large object data types (BLOB, CLOB, DBCLOB)” on page 191
- “Understanding large object locators” on page 191
- “Example: Using a locator to work with a CLOB value” on page 192
- “Indicator variables and LOB locators” on page 195
- “LOB file reference variables” on page 195

- “Example: Extracting a document to a file” on page 196
- “Example: Inserting data into a CLOB column” on page 198
- “Display layout of LOB columns” on page 199
- “Journal entry layout of LOB columns” on page 199

Understanding large object data types (BLOB, CLOB, DBCLOB)

Large object data types store data ranging in size from zero bytes to 2 gigabytes.

The three large object data types have the following definitions:

- Character Large Objects (CLOBs) — A character string made up of single-byte characters with an associated code page. This data type is best for holding text-oriented information where the amount of information could grow beyond the limits of a regular VARCHAR data type (upper limit of 32K bytes). Code page conversion of the information is supported as well as compatibility with the other character types.
- Double-Byte Character Large Objects (DBCLOBs) — A character string made up of double-byte characters with an associated code page. This data type is best for holding text-oriented information where double-byte character sets are used. Again, code page conversion of the information is supported as well as compatibility with the other double-byte character types.
- Binary Large Objects (BLOBs) — A binary string made up of bytes with no associated code page. This data type may be the most useful because it can store binary data. Therefore, it is a perfect source type for use by User-defined Distinct Types (UDTs). UDTs using BLOBs as the source type are created to store image, voice, graphical, and other types of business or application-specific data. For more information about UDTs, see “User-defined distinct types (UDT)” on page 215.

Understanding large object locators

Conceptually, LOB locators represent a simple idea that has been around for a while; use a small, easily managed value to refer to a much larger value. Specifically, a LOB locator is a 4-byte value stored in a host variable that a program uses to refer to a LOB value (or LOB expression) held in the database system. Using a LOB locator, a program can manipulate the LOB value as if the LOB value was stored in a regular host variable. When you use the LOB locator, there is no need to transport the LOB value from the server to the application (and possibly back again).

The LOB locator is associated with a LOB value or LOB expression, not a row or physical storage location in the database. Therefore, after selecting a LOB value into a locator, you cannot perform an operation on the original row(s) or tables(s) that would have any effect on the value referenced by the locator. The value associated with the locator is valid until the unit of work ends, or the locator is explicitly freed, whichever comes first. The FREE LOCATOR statement releases a locator from its associated value. In a similar way, a commit or roll-back operation frees all LOB locators associated with the transaction.

LOB locators can also be passed between DB2 and UDFs. Within the UDF, those functions that work on LOB data are available to manipulate the LOB values using LOB locators.

When selecting a LOB value, you have three options.

- Select the entire LOB value into a host variable. The entire LOB value is copied into the host variable.
- Select the LOB value into a LOB locator. The LOB value remains on the server; it is not copied to the host variable.
- Select the entire LOB value into a file reference variable. The LOB value is moved to an Integrated File System (IFS) file.

The use of the LOB value within the program can help the programmer to determine which method is best. If the LOB value is very large and is needed only as an input value for one or more subsequent SQL statements, keep the value in a locator.

If the program needs the entire LOB value regardless of the size, then there is no choice but to transfer the LOB. Even in this case, there are still options available to you. You can select the entire value into a regular or file reference host variable. You may also select the LOB value into a locator and read it piecemeal from the locator into a regular host variable, as suggested in the following example, "Example: Using a locator to work with a CLOB value".

Example: Using a locator to work with a CLOB value

In this example, the application program retrieves a locator for a LOB value; then it uses the locator to extract the data from the LOB value. Using this method, the program allocates only enough storage for one piece of LOB data (the size is determined by the program). In addition, the program needs to issue only one fetch call using the cursor.

Note: See "Code disclaimer information" on page x information for information pertaining to code examples.

How the sample LOBLOC program works

1. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. CLOB LOCATOR host variables are declared.
2. **Fetch the LOB value into the locator host variable.** A CURSOR and FETCH routine is used to obtain the location of a LOB field in the database to a locator host variable.
3. **Free the LOB LOCATORS.** The LOB LOCATORS used in this example are freed, releasing the locators from their previously associated values.

The CHECKERR macro/function is an error checking utility that is external to the program. The location of this error checking utility depends on the programming language that is used:

C check_error is redefined as CHECKERR and is located in the util.c file.

C Sample: LOBLOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {
```

```

#ifdef DB2MAC
    char * bufptr;
#endif

EXEC SQL BEGIN DECLARE SECTION; 1
    char number[7];
    long deptInfoBeginLoc;
    long deptInfoEndLoc;
    SQL TYPE IS CLOB_LOCATOR resume;
    SQL TYPE IS CLOB_LOCATOR deptBuffer;
    short lobind;
    char buffer[1000]="";
    char userid[9];
    char passwd[19];
EXEC SQL END DECLARE SECTION;

printf( "Sample C program: LOBLOC\n" );

if (argc == 1) {
    EXEC SQL CONNECT TO sample;
    CHECKERR ("CONNECT TO SAMPLE");
}
else if (argc == 3) {
    strcpy (userid, argv[1]);
    strcpy (passwd, argv[2]);
    EXEC SQL CONNECT TO sample USER :userid USING :passwd;
    CHECKERR ("CONNECT TO SAMPLE");
}
else {
    printf ("\nUSAGE: lobloc [userid passwd]\n\n");
    return 1;
} /* endif */

/* Employee A10030 is not included in the following select, because
the lobeval program manipulates the record for A10030 so that it is
not compatible with lobloc */

EXEC SQL DECLARE c1 CURSOR FOR
    SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
    AND empno <> 'A00130';

EXEC SQL OPEN c1;
CHECKERR ("OPEN CURSOR");

do {
    EXEC SQL FETCH c1 INTO :number, :resume :lobind; 2
    if (SQLCODE != 0) break;
    if (lobind < 0) {
        printf ("NULL LOB indicated\n");
    } else {
        /* EVALUATE the LOB LOCATOR */
        /* Locate the beginning of "Department Information" section */
        EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
            INTO :deptInfoBeginLoc;
        CHECKERR ("VALUES1");

        /* Locate the beginning of "Education" section (end of "Dept.Info" */
        EXEC SQL VALUES (POSSTR(:resume, 'Education'))
            INTO :deptInfoEndLoc;
        CHECKERR ("VALUES2");

        /* Obtain ONLY the "Department Information" section by using SUBSTR */
        EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
            :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
        CHECKERR ("VALUES3");

        /* Append the "Department Information" section to the :buffer var. */
        EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
        CHECKERR ("VALUES4");
    } /* endif */
} while ( 1 );

#ifdef DB2MAC
    /* Need to convert the newline character for the Mac */
    bufptr = &(buffer[0]);
    while ( *bufptr != '\0' ) {
        if ( *bufptr == 0x0A ) *bufptr = 0x0D;
        bufptr++;
    }
#endif

printf ("%s\n",buffer);

EXEC SQL FREE LOCATOR :resume, :deptBuffer; 3
CHECKERR ("FREE LOCATOR");

EXEC SQL CLOSE c1;
CHECKERR ("CLOSE CURSOR");

```

```

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : LOBLOC.SQC */

```

COBOL Sample: LOBLOC.SQB

```

Identification Division.
Program-ID. "lobloc".

Data Division.
Working-Storage Section.
  copy "sqlenv.cbl".
  copy "sql.cbl".
  copy "sqlca.cbl".

  EXEC SQL BEGIN DECLARE SECTION END-EXEC. 1
01 userid          pic x(8).
01 passwd.
  49 passwd-length pic s9(4) comp-5 value 0.
  49 passwd-name   pic x(18).
01 empnum         pic x(6).
01 di-begin-loc  pic s9(9) comp-5.
01 di-end-loc    pic s9(9) comp-5.
01 resume        USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 di-buffer     USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 lobind        pic s9(4) comp-5.
01 buffer        USAGE IS SQL TYPE IS CLOB(1K).
  EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc        pic x(80).

Procedure Division.
Main Section.
  display "Sample COBOL program: LOBLOC".

* Get database connection information.
  display "Enter your user id (default none): "
    with no advancing.
  accept userid.

  if userid = spaces
    EXEC SQL CONNECT TO sample END-EXEC
  else
    display "Enter your password : " with no advancing
    accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
  inspect passwd-name tallying passwd-length for characters
    before initial " ".

  EXEC SQL CONNECT TO sample USER :userid USING :passwd
    END-EXEC.
  move "CONNECT TO" to errloc.
  call "checkerr" using SQLCA errloc.

* Employee A10030 is not included in the following select, because
* the lobeval program manipulates the record for A10030 so that it is
* not compatible with lobloc

  EXEC SQL DECLARE c1 CURSOR FOR
    SELECT empno, resume FROM emp_resume
    WHERE resume_format = 'ascii'
    AND empno <> 'A00130' END-EXEC.

  EXEC SQL OPEN c1 END-EXEC.
  move "OPEN CURSOR" to errloc.
  call "checkerr" using SQLCA errloc.

  Move 0 to buffer-length.

  perform Fetch-Loop thru End-Fetch-Loop
    until SQLCODE not equal 0.

* display contents of the buffer.
  display buffer-data(1:buffer-length).

  EXEC SQL FREE LOCATOR :resume, :di-buffer END-EXEC. 3
  move "FREE LOCATOR" to errloc.
  call "checkerr" using SQLCA errloc.

  EXEC SQL CLOSE c1 END-EXEC.
  move "CLOSE CURSOR" to errloc.
  call "checkerr" using SQLCA errloc.

```



```

EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.
End-Main.
  go to End-Prog.

Fetch-Loop Section.
EXEC SQL FETCH c1 INTO :empnum, :resume :lobind 2
END-EXEC.

  if SQLCODE not equal 0
    go to End-Fetch-Loop.

* check to see if the host variable indicator returns NULL.
  if lobind less than 0 go to NULL-lob-indicated.

* Value exists. Evaluate the LOB locator.
* Locate the beginning of "Department Information" section.
EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
  INTO :di-begin-loc END-EXEC.
move "VALUES1" to errloc.
call "checkerr" using SQLCA errloc.

* Locate the beginning of "Education" section (end of Dept.Info)
EXEC SQL VALUES (POSSTR(:resume, 'Education'))
  INTO :di-end-loc END-EXEC.
move "VALUES2" to errloc.
call "checkerr" using SQLCA errloc.

  subtract di-begin-loc from di-end-loc.

* Obtain ONLY the "Department Information" section by using SUBSTR
EXEC SQL VALUES (SUBSTR(:resume, :di-begin-loc,
  :di-end-loc))
  INTO :di-buffer END-EXEC.
move "VALUES3" to errloc.
call "checkerr" using SQLCA errloc.

* Append the "Department Information" section to the :buffer var
EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
END-EXEC.
move "VALUES4" to errloc.
call "checkerr" using SQLCA errloc.

  go to End-Fetch-Loop.

NULL-lob-indicated.
  display "NULL LOB indicated".

End-Fetch-Loop. exit.

End-Prog.
  stop run.

```

Indicator variables and LOB locators

For normal host variables in an application program, when selecting a NULL value into a host variable, a negative value is assigned to the indicator variable signifying that the value is NULL. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Since a locator host variable itself can never be NULL, a negative indicator variable value indicates that the LOB value represented by the LOB locator is NULL. The NULL information is kept local to the client using the indicator variable value — the server does not track NULL values with valid locators.

LOB file reference variables

File reference variables are similar to host variables except that they are used to transfer data to and from IFS files (not to and from memory buffers). A file reference variable represents (rather than contains) the file, just as a LOB locator represents (rather than contains) the LOB value. Database queries, updates, and inserts may use file reference variables to store, or to retrieve, single LOB values.

For very large objects, files are natural containers. It is likely that most LOBs begin as data stored in files on the client before they are moved to the database on the

server. The use of file reference variables assists in moving LOB data. Programs use file reference variables to transfer LOB data from the IFS file directly to the database engine. To carry out the movement of LOB data, the application does not have to write utility routines to read and write files using host variables.

Note: The file referenced by the file reference variable must be accessible from (but not necessarily resident on) the system on which the program runs. For a stored procedure, this would be the server.

A file reference variable has a data type of BLOB, CLOB, or DBCLOB. It is used either as the source of data (input) or as the target of data (output). The file reference variable may have a relative file name or a complete path name of the file (the latter is advised). The file name length is specified within the application program. The data length portion of the file reference variable is unused during input. During output, the data length is set by the application requestor code to the length of the new data that is written to the file.

When using file reference variables there are different options on both input and output. You must choose an action for the file by setting the `file_options` field in the file reference variable structure. Choices for assignment to the field covering both input and output values are shown below.

Values (shown for C) and options when using input file reference variables are as follows:

- **SQL_FILE_READ** (Regular file) — This option has a value of 2. This is a file that can be open, read, and closed. DB2 determines the length of the data in the file (in bytes) when opening the file. DB2 then returns the length through the `data_length` field of the file reference variable structure. (The value for COBOL is SQL-FILE-READ.)

Values and options when using output file reference variables are as follows:

- **SQL_FILE_CREATE** (New file) — This option has a value of 8. This option creates a new file. Should the file already exist, an error message is returned. (The value for COBOL is SQL-FILE-CREATE.)
- **SQL_FILE_OVERWRITE** (Overwrite file) — This option has a value of 16. This option creates a new file if none already exists. If the file already exists, the new data overwrites the data in the file. (The value for COBOL is SQL-FILE-OVERWRITE.)
- **SQL_FILE_APPEND** (Append file) — This option has a value of 32. This option has the output appended to the file, if it exists. Otherwise, it creates a new file. (The value for COBOL is SQL-FILE-APPEND.)

Note: If a LOB file reference variable is used in an OPEN statement, do not delete the file associated with the LOB file reference variable until the cursor is closed.

For more information about integrated file system, see Integrated File System.

Example: Extracting a document to a file

This program example shows how CLOB elements can be retrieved from a table into an external file.

How the sample LOBFILE program works

1. **Declare host variables.** The BEGIN DECLARE SECTION and END DECLARE SECTION statements delimit the host variable declarations. Host variables are prefixed with a colon (:) when referenced in an SQL statement. A CLOB FILE REFERENCE host variable is declared.
2. **CLOB FILE REFERENCE host variable is set up.** The attributes of the FILE REFERENCE are set up. A file name without a fully declared path is, by default, placed in the user's current directory. If the pathname does not begin with the forward slash (/) character, it is not qualified.
3. **Select into the CLOB FILE REFERENCE host variable.** The data from the resume field is selected into the filename that is referenced by the host variable.

The CHECKERR macro/function is an error checking utility which is external to the program. The location of this error checking utility depends upon the programming language used:

C check_error is redefined as CHECKERR and is located in the util.c file.

COBOL CHECKERR is an external program named checkerr.cb1

Note: See "Code disclaimer information" on page x information for information pertaining to code examples.

C Sample: LOBFILE.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION; 1
    SQL TYPE IS CLOB_FILE resume;
    short lobind;
    char userid[9];
    char passwd[19];
    EXEC SQL END DECLARE SECTION;

    printf("Sample C program: LOBFILE\n");

    if (argc == 1) {
        EXEC SQL CONNECT TO sample;
        CHECKERR("CONNECT TO SAMPLE");
    }
    else if (argc == 3) {
        strcpy(userid, argv[1]);
        strcpy(passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd;
        CHECKERR("CONNECT TO SAMPLE");
    }
    else {
        printf("\nUSAGE: lobfile [userid passwd]\n\n");
        return 1;
    } /* endif */

    strcpy(resume.name, "RESUME.TXT"); 2
    resume.name_length = strlen("RESUME.TXT");
    resume.file_options = SQL_FILE_OVERWRITE;

    EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume 3
        WHERE resume_format='ascii' AND empno='000130';

    if (lobind < 0) {
        printf("NULL LOB indicated \n");
    } else {
        printf("Resume for EMPNO 000130 is in file : RESUME.TXT\n");
    } /* endif */
}
```

```

EXEC SQL CONNECT RESET;
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : LOBFILE.SQC */

```

COBOL Sample: LOBFILE.SQB

```

Identification Division.
Program-ID. "lobfile".

Data Division.
Working-Storage Section.
copy "sqlenv.cb1".
copy "sql.cb1".
copy "sqlca.cb1".

EXEC SQL BEGIN DECLARE SECTION END-EXEC. 1
01 userid          pic x(8).
01 passwd.
   49 passwd-length pic s9(4) comp-5 value 0.
   49 passwd-name   pic x(18).
01 resume         USAGE IS SQL TYPE IS CLOB-FILE.
01 lobind         pic s9(4) comp-5.
EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc         pic x(80).

Procedure Division.
Main Section.
   display "Sample COBOL program: LOBFILE".

* Get database connection information.
   display "Enter your user id (default none): "
   with no advancing.
   accept userid.

   if userid = spaces
      EXEC SQL CONNECT TO sample END-EXEC
   else
      display "Enter your password : " with no advancing
      accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
   inspect passwd-name tallying passwd-length for characters
   before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd
END-EXEC.
move "CONNECT TO" to errloc.
call "checkerr" using SQLCA errloc.

move "RESUME.TXT" to resume-NAME. 2
move 10 to resume-NAME-LENGTH.
move SQL-FILE-OVERWRITE to resume-FILE-OPTIONS.

EXEC SQL SELECT resume INTO :resume :lobind 3
FROM emp_resume
WHERE resume_format = 'ascii'
AND empno = '000130' END-EXEC.
if lobind less than 0 go to NULL-LOB-indicated.

display "Resume for EMPNO 000130 is in file : RESUME.TXT".
go to End-Main.

NULL-LOB-indicated.
display "NULL LOB indicated".

End-Main.
EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.
End-Prog.
stop run.

```

Example: Inserting data into a CLOB column

In the path description of the following C program segment:

- `userid` represents the directory for one of your users.
- `dirname` represents a subdirectory name of "userid".
- `filnam.1` can become the name of one of your documents that you wish to insert into the table.

- clobtab is the name of the table with the CLOB data type.

The following example shows how to insert data from a regular file referenced by :hv_text_file into a CLOB column:

```
strcpy(hv_text_file.name, "/home/userid/dirname/filnam.1");
hv_text_file.name_length = strlen("/home/userid/dirname/filnam.1");
hv_text_file.file_options = SQL_FILE_READ; /* this is a 'regular' file */

EXEC SQL INSERT INTO CLOBTAB
VALUES(:hv_text_file);
```

Display layout of LOB columns

When a row of data from a table holding LOB columns is displayed using CL commands such as Display Physical File Member (DSPPFM), the LOB data stored in that row will not be displayed. Instead, the Database will show a special value for the LOB columns. The layout of this special value is as follows:

- 13 to 28 bytes of hex zeros.
- 16 bytes beginning with *POINTER and followed by blanks.

The number of bytes in the first portion of the value is set to the number needed to 16 byte boundary align the second part of the value.

For example, say you have a table that holds three columns: ColumnOne Char(10), ColumnTwo CLOB(40K), and ColumnThree BLOB(10M). If you were to issue a DSPPFM of this table, each row of data would look as follows.

- For ColumnOne: 10 bytes filled with character data.
- For ColumnTwo: 22 bytes filled with hex zeros and 16 bytes filled with *POINTER '.
- For ColumnThree: 16 bytes filled with hex zeros and 16 bytes filled with *POINTER '.

The full set of commands that display LOB columns in this way is:

- Display Physical File Member (DSPPFM)
- Copy File (CPYF) when the value *PRINT is specified for the TOFILE keyword
- Display Journal (DSPJRN)
- Retrieve Journal Entry (RTVJRNE)
- Receive Journal Entry (RCVJRNE) when the values *TYPE1, *TYPE2, *TYPE3 and *TYPE4 are specified for the ENTFMT keyword.

Journal entry layout of LOB columns

Two commands return a buffer that gives the user addressability to LOB data that had been journaled:

- Receive Journal Entry (RCVJRNE) CL command, when the value *TYPEPTR is specified for the ENTFMT keyword
- Retrieve Journal Entries (QjoRetrieveJournalEntries) API

The layout of the LOB columns in these entries is as follows:

- 0 to 15 bytes of hex zeros
- 1 byte of system information set to '00'x
- 4 bytes holding the length of the LOB data addressed by the pointer, below
- 8 bytes of hex zeros
- 16 bytes holding a pointer to the LOB data stored in the Journal Entry.

The first part of this layout is intended to 16 byte boundary align the pointer to the LOB data. The number of bytes in this area depends on the length of the columns that proceed the LOB column. Refer to the section above on the Display Layout of LOB Columns for an example of how the length of this first part is calculated.

For more information about the Journal handling of LOB columns, refer to the Journaling topic.

User-defined functions (UDF)

A user-defined function is a mechanism with which you can write your own extensions to SQL. The built-in functions supplied with DB2 are a useful set of functions, but they may not satisfy all of your requirements. Thus, you may need to extend SQL for the following reasons:

- **Customization.**

The function specific to your application does not exist in DB2. Whether the function is a simple transformation, a trivial calculation, or a complicated multivariate analysis, you can probably use a UDF to do the job.

- **Flexibility.**

The DB2 built-in function does not quite permit the variations that you wish to include in your application.

- **Standardization.**

Many of the programs at your site implement the same basic set of functions, but there are minor differences in all the implementations. Thus, you are unsure about the consistency of the results you receive. If you correctly implement these functions once, in a UDF, then all these programs can use the same implementation directly in SQL and provide consistent results.

- **Object-relational support.**

As discussed in “User-defined distinct types (UDT)” on page 215, UDTs can be very useful in extending the capability and increasing the safety of DB2. UDFs act as the *methods* for UDTs, by providing behavior and encapsulating the types.

In addition, SQL UDFs provide the support to manipulate Large Objects and DataLink types. While the database provides several built-in functions that are useful in working with these datatypes, SQL UDFs provide a way for users to further manipulate and enhance the capabilities of the database (to the specialization required) in this area.

For more details, see the following sections:

- “Why use UDFs?”
- “UDF concepts” on page 203
- “Implementing UDFs” on page 205
- “Registering UDFs” on page 205
- “Save and restore considerations” on page 206
- “Examples: Registering UDFs” on page 206
- “Using UDFs” on page 210

Why use UDFs?

In writing DB2 applications, you have a choice when implementing desired actions or operations:

- As a UDF

- As a subroutine or function in your application.

Although it appears easier to implement new operations as subroutines or functions in your application, you should still consider:

- **Re-use.**

If the new operation is something that other users or programs at your site can take advantage of, then UDFs can help to reuse it. In addition, the function can be invoked directly in SQL wherever an expression can be used by any user of the database. The database takes care of many data type promotions of the function arguments automatically. For example, with DECIMAL to DOUBLE, the database allows your function to be applied to different, but compatible data types.

It may seem easier to implement your new function as a normal function. (You would not have to define the function to DB2.) If you did this, you would have to inform all other interested application developers, and package the function effectively for their use. However, this process ignores the interactive users such as those who normally use the Command Line Processor (CLP) to access the database. However, functions written for use only within programs ignores those (interactive) users who do not have associated programs. This includes commands such as STRSQL, STRQM, and RUNSQLSTM, in addition to many clients such as ODBC, JDBC, etc. CLP users cannot use your function unless it is a UDF in the database. This also applies to any other tools that use SQL (such as Visualizer), that do not get recompiled.

- **Performance.**

In certain cases, invoking the UDF directly from the database engine instead of from your application can have a considerable performance advantage. You will notice this advantage in cases where the function may be used in the qualification of data for further processing. These cases occur when the function is used in row selection processing.

Consider a simple scenario where you want to process some data. You can meet some selection criteria which can be expressed as a function `SELECTION_CRITERIA()`. Your application could issue the following select statement:

```
SELECT A, B, C FROM T
```

When it receives each row, it runs `SELECTION_CRITERIA` against the data to decide if it is interested in processing the data further. Here, every row of table `T` must be passed back to the application. But, if `SELECTION_CRITERIA()` is implemented as a UDF, your application can issue the following statement:

```
SELECT C FROM T WHERE SELECTION_CRITERIA(A,B)=1
```

In this case, only the rows and column of interest are passed across the interface between the application and the database.

Another case where a UDF can offer a performance benefit is when dealing with Large Objects (LOB). Suppose you have a function that extracts some information from a value of one of the LOB types. You can perform this extraction right on the database server and pass only the extracted value back to the application. This is more efficient than passing the entire LOB value back to the application and then performing the extraction. The performance value of packaging this function as a UDF could be enormous, depending on the particular situation. (Note that you can also extract a portion of a LOB by using a LOB locator. See "Indicator variables and LOB locators" on page 195 for an example of a similar scenario.)

- **Object Orientation.**

You can implement the behavior of a user-defined distinct type (UDT), also called *distinct type*, using a UDF. For more information about UDTs, see “User-defined distinct types (UDT)” on page 215. For additional details on UDTs and the important concept of *castability* discussed herein, see the CREATE DISTINCT TYPE statement in the SQL Reference. When you create a distinct type, you are automatically provided cast functions between the distinct type and its source type. You may also be provided comparison operators, such as =, >, <, and so on, depending on the source type. You have to provide any additional behavior yourself. It is best to keep the behavior of a distinct type in the database where all of the users of the distinct type can easily access it. You can use UDFs, therefore, as the implementation mechanism.

For example, suppose that you have a BOAT distinct type, defined over a one megabyte BLOB. The type create statement:

```
CREATE DISTINCT TYPE BOAT AS BLOB(1M)
```

The BLOB contains the various nautical specifications, and some drawings. You may wish to compare sizes of boats. However, with a distinct type defined over a BLOB source type, you do not get the comparison operations automatically generated for you. You can implement a BOAT_COMPARE function that decides if one boat is bigger than another based on a measurement that you choose. These could be: displacement, length over all, metric tonnage, or another calculation based on the BOAT object. You create the BOAT_COMPARE function as follows:

```
CREATE SQL FUNCTION BOAT_COMPARE (BOAT, BOAT) RETURNS INTEGER ...
```

If your function returns:

- 1 the first BOAT is bigger
- 2 the second is bigger and
- 0 they are equal.

You could use this function in your SQL code to compare boats. Suppose you create the following tables:

```
CREATE TABLE BOATS_INVENTORY (  

  BOAT_ID      CHAR(5),  

  BOAT_TYPE    VARCHAR(25),  

  DESIGNER     VARCHAR(40),  

  OWNER        VARCHAR(40),  

  DESIGN_DATE  DATE,  

  SPEC         BOAT,  

  ...         )
```

```
CREATE TABLE MY_BOATS (  

  BOAT_ID      CHAR(5),  

  BOAT_TYPE    VARCHAR(25),  

  DESIGNER     VARCHAR(40),  

  DESIGN_DATE  DATE,  

  ACQUIRE_DATE DATE,  

  ACQUIRE_PRICE CANADIAN_DOLLAR,  

  CURR_APPRAISL CANADIAN_DOLLAR,  

  SPEC         BOAT,  

  ...         )
```

You can execute the following SQL SELECT statement:


```

SELECT INV.BOAT_ID, INV.BOAT_TYPE, INV.DESIGNER,
       INV.OWNER, INV.DESIGN_DATE
FROM BOATS_INVENTORY INV, MY_BOATS MY
WHERE MY.BOAT_ID = '19GCC'
AND BOAT_COMPARE(INV.SPEC, MY.SPEC) = 1
AND INV.DESIGNER = MY.DESIGNER

```

This simple example returns all the boats from BOATS_INVENTORY from the same designer that are bigger than a particular boat in MY_BOATS. Note that the example only passes the rows of interest back to the application because the comparison occurs in the database server. In fact, it completely avoids passing any values of data type BOAT. This is a significant improvement in storage and performance as BOAT is based on a one megabyte BLOB data type.

UDF concepts

The following is a discussion of the important concepts you need to know prior to coding UDFs:

Function Name

- Full name of a function.

The full name of a function using *SQL naming is <schema-name>.<function-name>.

The full name of a function in *SYS naming is <schema-name>/<function-name>. Function names cannot be qualified using *SYS naming in DML statements.

You can use this full name anywhere you refer to a function. For example:

```

QGPL.SNOWBLOWER_SIZE    SMITH.FOO    QSYS2.SUBSTR    QSYS2.FLOOR

```

However, you may also omit the <schema-name>., in which case, DB2 must determine the function to which you are referring. For example:

```

SNOWBLOWER_SIZE    FOO    SUBSTR    FLOOR

```

- Path

The concept of *path* is central to DB2's resolution of *unqualified* references that occur when *schema-name* is not specified. For the use of *path* in DDL statements that refer to functions, see the description of the corresponding CREATE FUNCTION statement in the SQL Reference. The path is an ordered list of schema names. It provides a set of schemas for resolving unqualified references to UDFs as well as UDTs. In cases where a function reference matches functions in more than one schema in the path, the order of the schemas in the path is used to resolve this match. The path is established by means of the SQLPATH option on the precompile and bind commands for static SQL. The path is set by the SET PATH statement for dynamic SQL. When the first SQL statement that runs in an activation group runs with SQL naming, the path has the following default value:

```
"QSYS", "QSYS2", "<ID>"
```

This applies to both static and dynamic SQL, where <ID> represents the current statement authorization ID.

When the first SQL statement in an activation group runs with system naming, the default path is *LIBL.

- Overloaded function names.

Function names can be *overloaded*. Overloaded means that multiple functions, even in the same schema, can have the same name. Two functions cannot, however, have the same *signature*. A function signature can be defined to be the

qualified function name concatenated with the defined data types of all the function parameters in the order in that they are defined. For an example of an overloaded function, see “Example: BLOB string search” on page 207. See the Function topic in the SQL Reference book for more information about signature and function resolution.

- Function resolution.

It is the *function resolution algorithm* that takes into account the facts of overloading and function path to choose the *best fit* for every function reference, whether it is a qualified or an unqualified reference. All functions, even built-in functions, are processed through the function selection algorithm. The function resolution algorithm does not take into account the type of a function. So a table function may be resolved to as the *best fit* function, even though the usage of the reference would require an scalar function, or vice-versa.

The concept of path, the SET PATH statement, and the function resolution algorithm are discussed in detail in the SQL Reference. The SQLPATH precompile option is discussed in the command appendix.

- Types of function.

There are several types of functions:

- *Built-in*. These are functions provided by and shipped with the database. SUBSTR() is an example.
- *System-generated*. These are functions implicitly generated by the database engine when a DISTINCT TYPE is created. These functions provide casting operations between the DISTINCT TYPE and its base type.
- *User-defined*. These are functions created by users and registered to the database.

In addition, each function can be further classified as a *scalar*, *column*, or *table* function.

A *scalar function* returns a single value answer each time it is called. For example, the built-in function SUBSTR() is a scalar function, as are many built-in functions. System-generated functions are always scalar functions. Scalar UDFs can either be external (coded in a programming language such as C, or in SQL—an SQL function), or sourced (using the implementation of an existing function).

A *column function* receives a set of like values (a column of data) and returns a single value answer from this set of values. These are also called *aggregating functions* in DB2. Some built-in functions are column functions. An example of a column function is the built-in function AVG(). An external UDF cannot be defined as a column function. However, a sourced UDF is defined to be a column function if it is sourced on one of the built-in column functions. The latter is useful for distinct types. For example, if a distinct type SHOESIZE exists that is defined with base type INTEGER, you could define a UDF, AVG(SHOESIZE), as a column function sourced on the existing built-in column function, AVG(INTEGER).

A *table function* returns a table to the SQL statement that references it. It must be referenced in the FROM clause of a SELECT. A table function can be used to apply SQL language processing power to data that is not DB2 data, or to convert such data into a DB2 table. It could, for example, take a file and convert it to a table, sample data from the World Wide Web and tabularize it, or access a Lotus Notes database and return information about mail messages, such as the date, sender, and the text of the message. This information can be joined with other

tables in the database. A table function can be defined as a external function or an SQL function; it cannot be defined as a sourced function.

Implementing UDFs

There are three types of UDFs: sourced, external, and SQL. The implementation of each type is considerably different.

- Sourced UDFs. These are simply functions registered to the database that themselves reference another function. They, in effect, map the sourced function. As such, nothing more is required in implementing these functions than registering them to the database using the CREATE FUNCTION statement.
- External functions. These are references to programs and service programs written in a high level language such as C, COBOL, or RPG. Once the function is registered to the database, the database will invoke the program or service program whenever the function is referenced in a DML statement. As such, external UDFs require that the UDF writer, besides knowing the high level language and how to develop code in it, understand the interface between the program and the database. See Chapter 13, “Writing User-Defined Functions (UDFs)” on page 231 for more information about writing external functions.
- SQL UDFs. SQL UDFs are functions written entirely in the SQL language. Their ‘code’ is actually SQL statements embedded within the CREATE FUNCTION statement itself. SQL UDFs provide several advantages:
 - They are written in SQL, making them quite portable.
 - Defining the interface between the database and the function is by use of SQL declares, with no need to worry about details of actual parameter passing.
 - They allow the passing of large objects, datalinks, and UDTs as parameters, and subsequent manipulation of them in the function itself. More information about SQL functions can be found in Chapter 13, “Writing User-Defined Functions (UDFs)” on page 231.

To use a UDF, take the following steps:

1. Registering the UDF with DB2. Regardless of which type of UDF is being created, they all need to be registered to the database using the CREATE FUNCTION statement. In the case of source functions, this registration step does everything necessary to define the function to the database. For SQL UDFs, the CREATE FUNCTION statement contains everything necessary to define the function as well, except that the syntax of the CREATE statement is much more complex (contains actual SQL executable code). For external UDFs, the CREATE FUNCTION statement **only** registers the function to the database; the supporting code that actually implements the function must be written separately. See “Registering UDFs” for more information.
2. Debugging the UDF. See Chapter 13, “Writing User-Defined Functions (UDFs)” on page 231.

After these steps are successfully completed, your UDF is ready for use in data manipulation language (DML) or data definition language (DDL) statements such as CREATE VIEW.

Registering UDFs

A UDF must be registered in the database before the function can be recognized and used by the database. You can register a UDF using the CREATE FUNCTION statement.

The statement allows you to specify the language and name of the program, along with options such as DETERMINISTIC, ALLOW PARALLEL, and RETURNS NULL ON NULL INPUT. These options help to more specifically identify to the database the intention of the function and how calls to the database can be optimized.

You should register an external UDF to DB2 after you have written and completely tested the actual code. It is possible to define the UDF prior to actually writing it. However, to avoid any problems with running your UDF, you are encouraged to write and test it extensively before registering it. For information on testing your UDF, see Chapter 13, “Writing User-Defined Functions (UDFs)” on page 231.

For an example of registering UDFs, see “Examples: Registering UDFs”.

Save and restore considerations

When an external function associated with an ILE external program or service program is created, an attempt is made to save the function’s attributes in the associated program or service program object. If the *PGM or *SRVPGM object is saved and then restored to this or another system, the catalogs are automatically updated with those attributes. If the the function’s attribute could not be saved, then the catalogs will not be automatically updated and the user must create the external function on the new system. The attributes can be saved for external functions subject to the following restrictions:

- The external program library must not be QSYS or QSYS2.
- The external program must exist when the CREATE FUNCTION statement is issued.
- The external program must be an ILE *PGM or *SRVPGM object.
- The external program or service program must contain at least one SQL statement.

If the object cannot be updated, the function will still be created.

Examples: Registering UDFs

The examples which follow illustrate a variety of typical situations where UDFs can be registered. The examples include:

- Example: Exponentiation
- Example: String search
- Example: String search over UDT
- Example: External function with UDT parameter
- Example: AVG over a UDT
- Example: Counting
- Example: Table function returning Document IDs

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Example: Exponentiation

Suppose you have written an external UDF to perform exponentiation of floating point values, and wish to register it in the MATH schema.

```
CREATE FUNCTION MATH.EXPON (DOUBLE, DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/MYPGM(MYENTRY) '
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
```

```
DETERMINISTIC
NO EXTERNAL ACTION
RETURNS NULL ON NULL INPUT
ALLOW PARALLEL
```

In this example, the system uses the RETURNS NULL ON NULL INPUT default value. This is desirable since you want the result to be NULL if either argument is NULL. Since you do not require a scratchpad and no final call is necessary, the NO SCRATCHPAD and NO FINAL CALL default values are used. As there is no reason why EXPON cannot be parallel, the ALLOW PARALLEL value is specified.

Example: String search

Your associate, Willie, has written a UDF to look for the existence of a given short string, passed as an argument, within a given CLOB value, that is also passed as an argument. The UDF returns the position of the string within the CLOB if it finds the string, or zero if it does not.

Additionally, Willie has written the function to return a FLOAT result. Suppose you know that when it is used in SQL, it should always return an INTEGER. You can create the following function:

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
RETURNS INTEGER
CAST FROM FLOAT
SPECIFIC "willie_find_feb95"
EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION
RETURNS NULL ON NULL INPUT
```

Note that a CAST FROM clause is used to specify that the UDF body really returns a FLOAT value, but you want to cast this to INTEGER before returning the value to the statement which used the UDF. As discussed in the SQL Reference, the INTEGER built-in function can perform this cast for you. Also, you wish to provide your own specific name for the function and later reference it in DDL (see "Example: String search over UDT" on page 208). Because the UDF was not written to handle NULL values, you use the RETURNS NULL ON NULL INPUT. And because there is no scratchpad, you use the NO SCRATCHPAD and NO FINAL CALL default values. As there is no reason why FINDSTRING cannot be parallel, the ALLOW PARALLEL default value is used.

Example: BLOB string search

Because you want this function to work on BLOBs as well as on CLOBs, you define another FINDSTRING taking BLOB as the first parameter:

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
RETURNS INTEGER
CAST FROM FLOAT
SPECIFIC "willie_fblob_feb95"
EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION
```

This example illustrates overloading of the UDF name and shows that multiple UDFs can share the same body. Note that although a BLOB cannot be assigned to a CLOB, the same source code can be used. There is no programming problem in the

above example as the programming interface for BLOB and CLOB between DB2 and UDF is the same: length followed by data. DB2 does not check if the UDF using a particular function body is in any way consistent with any other UDF using the same body.

Example: String search over UDT

This example is a continuation of the previous example. Say you are satisfied with the FINDSTRING functions from “Example: BLOB string search” on page 207, but now you have defined a distinct type BOAT with source type BLOB. You also want FINDSTRING to operate on values having data type BOAT, so you create another FINDSTRING function. This function is sourced on the FINDSTRING which operates on BLOB values in “Example: BLOB string search” on page 207. Note the further overloading of FINDSTRING in this example:

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INT
  SPECIFIC "slick_boat_mar95"
  SOURCE SPECIFIC "willie_fblob_feb95"
```

Note that this FINDSTRING function has a different signature from the FINDSTRING functions in “Example: BLOB string search” on page 207, so there is no problem overloading the name. You wish to provide your own specific name for possible later reference in DDL. Because you are using the SOURCE clause, you cannot use the EXTERNAL NAME clause or any of the related keywords specifying function attributes. These attributes are taken from the source function. Finally, observe that in identifying the source function you are using the specific function name explicitly provided in “Example: BLOB string search” on page 207. Because this is an unqualified reference, the schema in which this source function resides must be in the function path, or the reference will not be resolved.

Example: External function with UDT parameter

You have written another UDF to take a BOAT and examine its design attributes and generate a cost for the boat in Canadian dollars. Even though internally, the labor cost may be priced in German marks, or Japanese yen, or US dollars, this function needs to generate the cost to build the boat in the required currency: Canadian dollars. This means that the function has to get current exchange rate information from the exchange_rate file, managed outside of DB2, and the answer depends on what the function finds in this file. This makes the function NOT DETERMINISTIC.

```
CREATE FUNCTION BOAT_COST (BOAT)
  RETURNS INTEGER
  EXTERNAL NAME 'MYLIB/COSTS(BOATCOST)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
```

Observe that CAST FROM and SPECIFIC are not specified, but that NOT DETERMINISTIC is specified.

Example: AVG over a UDT

This example implements the AVG column function over the CANADIAN_DOLLAR distinct type. See “Example: Money” on page 217 for the definition of CANADIAN_DOLLAR. Strong typing prevents you from using the built-in AVG function on a distinct type. It turns out that the source type for CANADIAN_DOLLAR was DECIMAL, and so you implement the AVG by sourcing it on the AVG(DECIMAL) built-in function. The ability to do this depends

on being able to cast from DECIMAL to CANADIAN_DOLLAR and vice versa, but since DECIMAL is the source type for CANADIAN_DOLLAR, you know these casts will work.

```
CREATE FUNCTION AVG (CANADIAN_DOLLAR)
RETURNS CANADIAN_DOLLAR
SOURCE "QSYS2".AVG(DECIMAL(9,2))
```

Note that in the SOURCE clause you have qualified the function name, just in case there might be some other AVG function lurking in your SQL path.

Example: Counting

Your simple counting function returns a 1 the first time and increments the result by one each time it is called. This function takes no SQL arguments, and by definition it is a NOT DETERMINISTIC function since its answer varies from call to call. It uses the SCRATCHPAD to save the last value returned. Each time it is invoked the function increments this value and returns it.

```
CREATE FUNCTION COUNTER ()
RETURNS INT
EXTERNAL NAME 'MYLIB/MYFUNCS(CTR)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
NOT DETERMINISTIC
NOT FENCED
SCRATCHPAD 4
DISALLOW PARALLEL
```

Note that no parameter definitions are provided, just empty parentheses. The above function specifies SCRATCHPAD and uses the default specification of NO FINAL CALL. In this case, the size of the scratchpad is set to only 4 bytes, which is sufficient for a counter. Since the COUNTER function requires that a single scratchpad be used to operate properly, DISALLOW PARALLEL is added to prevent DB2 from operating it in parallel.

Example: Table function returning Document IDs

You have written a table function that returns a row consisting of a single document identifier column for each known document in your text management system that matches a given subject area (the first parameter) and contains the given string (second parameter). This UDF uses the functions of the text management system to quickly identify the documents:

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
RETURNS TABLE (DOC_ID CHAR(16))
EXTERNAL NAME 'DOCFUNCS/UDFMATCH(udfmatch)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION
NOT FENCED
SCRATCHPAD
NO FINAL CALL
DISALLOW PARALLEL
CARDINALITY 20
```

Within the context of a single session it will always return the same table, and therefore it is defined as DETERMINISTIC. Note the RETURNS clause which defines the output from DOCMATCH, including the column name DOC_ID. FINAL CALL does not need to be specified for each table function. In addition, the DISALLOW PARALLEL keyword is added as table functions cannot operate in

parallel. Although the size of the output from DOCMATCH is high variable, CARDINALITY 20 is a representative value, and is specified to help the DB2 optimizer to make good decisions.

Typically, this table function would be used in a join with the table containing the document text, as follows:

```
SELECT T.AUTHOR, T.DOCTEXT
FROM DOCS AS T, TABLE(DOCMATCH('MATHEMATICS', 'ZORN'S LEMMA')) AS F
WHERE T.DOCID = F.DOC_ID
```

Note the special syntax (TABLE keyword) for specifying a table function in a FROM clause. In this invocation, the DOCMATCH() table function returns a row containing the single column DOC_ID for each MATHEMATICS document referencing ZORN'S LEMMA. These DOC_ID values are joined to the master document table, retrieving the author's name and document text.

Using UDFs

Scalar and column UDFs can be invoked within an SQL statement almost everywhere that an expression is valid. Table UDFs can be invoked in the FROM clause of a SELECT. There are a few restrictions of UDF usage, however:

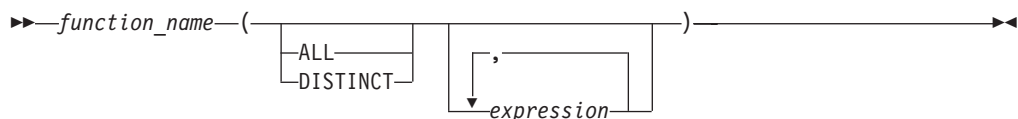
- UDFs and system generated functions cannot be specified in check constraints. Check constraints also cannot contain references to the built-in functions DLVALUE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, DLURLCOMPLETE, DLURLSERVER, ATAN2, DIFFERENCE, RADIANS, RAND, SOUNDEX, NOW, CURDATE, and CURTIME .
- External UDFs, SQL UDFs and the built-in functions DLVALUE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, DLURLCOMPLETE, and DLURLSERVER cannot be referenced in an ORDER BY or GROUP BY clause, unless the SQL statement is read-only and allows temporary processing (ALWCOPYDATA(*YES) or (*OPTIMIZE)).

The SQL Reference discusses all these contexts in detail. The discussion and examples used in this section focus on relatively simple SELECT statement contexts, however, note that their use is not restricted to these contexts.

Refer to "UDF concepts" on page 203 for a summary of the use and importance of the *path* and the *function resolution* algorithm. You can find the details for both of these concepts in the SQL Reference. The resolution of any Data Manipulation Language (DML) reference to a function uses the function resolution algorithm, so it is important to understand how it works.

Referring to functions

Each reference to a function, whether it is a UDF, or a built-in function, contains the following syntax:



In the above, `function_name` can be either an unqualified or a qualified function name. Note that when using the *SYS naming convention, functions cannot be qualified. The arguments can number from 0 to 90, and are expressions that may contain:

- A column name, qualified or unqualified

- A constant
- An expression
- A function
- A host variable
- A parameter marker with a CAST function
- A scalar-subselect
- A special register

The position of the arguments is important and must conform to the function definition for the semantics to be correct. Both the position of the arguments and the function definition must conform to the function body itself. DB2 does not attempt to shuffle arguments to better match a function definition, and DB2 does not attempt to determine the semantics of the individual function parameters.

Examples of function invocations

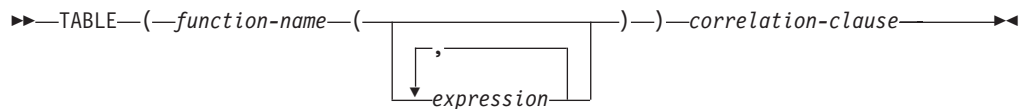
Some valid examples of function invocations are:

```

AVG(FLOAT_COLUMN)
BLOOP(COLUMN1)
BLOOP(FLOAT_COLUMN + CAST(? AS INTEGER))
BLOOP(:hostvar :indicvar)
BRIAN.PARSE(CONCAT(CHAR_COLUMN,USER), 1, 0, 0, 1)
CTR()
FLOOR(FLOAT_COLUMN)
PABLO.BLOOP(A+B)
PABLO.BLOOP(:hostvar)
"search_schema"(CURRENT PATH, 'GENE')
SUBSTR(COLUMN2,8,3)
QSYS2.FLOOR(AVG(EMP.SALARY))
QSYS2.AVG(QSYS2.FLOOR(EMP.SALARY))
QSYS2.SUBSTR(COLUMN2,11,LENGTH(COLUMN3))

```

Referring to table functions



In the above diagram, `function_name` can be either an unqualified or a qualified function name. Note that user-defined table functions can be qualified when using either the `*SYS` or `*SQL` naming convention.

Some valid examples of table function invocations are:

```

TABLE(TABFUNC()) X
TABLE(BLOOP_TAB(:hostvar, COL1+COL2))
NEW_TABLE TABLE(SCHEMA1.BLOOP_TAB2()) X

```

Using parameter markers or NULL in functions

An important restriction involves both parameter markers and the NULL value; you cannot simply code the following:

```
BLOOP(?)
```

or

```
BLOOP(NULL)
```

Since function resolution does not know what data type the argument may turn out to be, it cannot resolve the reference. You can use the CAST specification to provide a type for the parameter marker or NULL value, for example INTEGER, that function resolution can use:

```
BLOOP(CAST(? AS INTEGER))
```

or

```
BLOOP(CAST(NULL AS INTEGER))
```

Using qualified function reference

If you use a qualified function reference, you restrict DB2's search for a matching function to that schema. For example, you have the following statement:

```
SELECT PABLO.BLOOP(COLUMN1) FROM T
```

Only the BLOOP functions in schema PABLO are considered. It does not matter that user SERGE has defined a BLOOP function, or whether or not there is a built-in BLOOP function. Now suppose that user PABLO has defined two BLOOP functions in his schema:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS ...  
CREATE FUNCTION BLOOP (DOUBLE) RETURNS ...
```

BLOOP is thus overloaded within the PABLO schema, and the function selection algorithm would choose the best BLOOP, depending on the data type of the argument, column1. In this case, both of the PABLO.BLOOPS take numeric arguments, and if column1 is not one of the numeric types, the statement will fail. On the other hand if column1 is either SMALLINT or INTEGER, function selection will resolve to the first BLOOP, while if column1 is DECIMAL or DOUBLE, the second BLOOP will be chosen.

Several points about this example:

1. It illustrates argument promotion. The first BLOOP is defined with an INTEGER parameter, yet you can pass it a SMALLINT argument. The function selection algorithm supports promotions among the built-in data types (for details, see the SQL Reference) and DB2 performs the appropriate data value conversions.
2. If for some reason you want to invoke the second BLOOP with a SMALLINT or INTEGER argument, you have to take an explicit action in your statement as follows:

```
SELECT PABLO.BLOOP(DOUBLE(COLUMN1)) FROM T
```

3. Alternatively, if you want to invoke the first BLOOP with a DECIMAL or DOUBLE argument, you have your choice of explicit actions, depending on your exact intent:

```
SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T  
SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
```

You should investigate these other functions in the SQL Reference. The INTEGER function is a built-in function in the QSYS2 schema.

Using unqualified function reference

If, instead of a qualified function reference, you use an unqualified function reference, DB2's search for a matching function normally uses the function path to qualify the reference. In the case of the DROP FUNCTION or COMMENT ON FUNCTION functions, the reference is qualified using the current authorization ID, if they are unqualified for *SQL naming, or *LIBL for *SYS naming. Thus, *it is important that you know what your function path is, and what, if any, conflicting functions exist in the schemas of your current function path.* For example, suppose you are PABLO and your static SQL statement is as follows, where COLUMN1 is data type INTEGER:

```
SELECT BLOOP(COLUMN1) FROM T
```

You have created the two BLOOP functions cited in “Using qualified function reference” on page 212, and you want and expect one of them to be chosen. If the following default function path is used, the first BLOOP is chosen (since column1 is INTEGER), if there is no conflicting BLOOP in QSYS or QSYS2:

```
"QSYS", "QSYS2", "PABLO"
```

However, suppose you have forgotten that you are using a script for precompiling and binding which you previously wrote for another purpose. In this script, you explicitly coded your SQLPATH parameter to specify the following function path for another reason that does not apply to your current work:

```
"KATHY", "QSYS", "QSYS2", "PABLO"
```

If Kathy has written a BLOOP function for her own purposes, the function selection could very well resolve to Kathy’s function, and your statement would execute without error. You are not notified because DB2 assumes that you know what you are doing. It becomes your responsibility to identify the incorrect output from your statement and make the required correction.

Summary of function references

For both qualified and unqualified function references, the function selection algorithm looks at all the applicable functions, both built-in and user-defined, that have:

- The given name
- The same number of defined parameters as arguments in the function reference
- Each parameter identical to or promotable from the type of the corresponding argument.

(*Applicable functions* means *functions in the named schema* for a qualified reference, or *functions in the schemas of the function path* for an unqualified reference.) The algorithm looks for an exact match, or failing that, a best match among these functions. The current function path is used, in the case of an unqualified reference only, as the deciding factor if two identically good matches are found in different schemas. The details of the algorithm can be found in the SQL Reference.

An interesting feature, illustrated by the examples at the end of “Using qualified function reference” on page 212, is *the fact that function references can be nested*, even references to the same function. This is generally true for built-in functions as well as UDFs; however, there are some limitations when column functions are involved.

Refining an earlier example:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

Now consider the following DML statement:

```
SELECT BLOOP( BLOOP(COLUMN1)) FROM T
```

If column1 is a DECIMAL or DOUBLE column, the inner BLOOP reference resolves to the second BLOOP defined above. Because this BLOOP returns an INTEGER, the outer BLOOP resolves to the first BLOOP.

Alternatively, if column1 is a SMALLINT or INTEGER column, the inner bloop reference resolves to the first BLOOP defined above. Because this BLOOP returns an INTEGER, the outer BLOOP also resolves to the first BLOOP. In this case, you are seeing nested references to the same function.

A few additional points important for function references are:

- You can define a function with the name of one of the SQL operators. For example, suppose you can attach some meaning to the "+" operator for values which have distinct type BOAT. You can define the following UDF:

```
CREATE FUNCTION "+" (BOAT, BOAT) RETURNS ...
```

Then you can write the following valid SQL statement:

```
SELECT "+"(BOAT_COL1, BOAT_COL2)
FROM BIG_BOATS
WHERE BOAT_OWNER = 'Nelson Mattos'
```

Note that you are not permitted to overload the built-in conditional operators such as >, =, LIKE, IN, and so on, in this way.

- The function selection algorithm does not consider the context of the reference in resolving to a particular function. Look at these BLOOP functions, modified a bit from before:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS CHAR(10)...
```

Now suppose you write the following SELECT statement:

```
SELECT 'ABCDEFGF' CONCAT BLOOP(SMALLINT_COL) FROM T
```

Because the best match, resolved using the SMALLINT argument, is the first BLOOP defined above, the second operand of the CONCAT resolves to data type INTEGER. The statement fails because CONCAT demands string arguments. If the first BLOOP was not present, the other BLOOP would be chosen and the statement execution would be successful.

- UDFs can be defined with parameters or results having any of the LOB types: BLOB, CLOB, or DBCLOB. DB2 will materialize the entire LOB value in storage before invoking such a function, even if the source of the value is a *LOB locator* host variable. For example, consider the following fragment of a C language application:

```
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS CLOB(150K) clob150K ;      /* LOB host var */
SQL TYPE IS CLOB_LOCATOR clob_locator1; /* LOB locator host var */
char string[40]; /* string host var */
EXEC SQL END DECLARE SECTION;
```

Either host variable :clob150K or :clob_locator1 is valid as an argument for a function whose corresponding parameter is defined as CLOB(500K). Thus, referring to the FINDSTRING defined in "Example: String search" on page 207, both of the following are valid in the program:

```
... SELECT FINDSTRING (:clob150K, :string) FROM ...
... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
```

- Non-SQL UDF parameters or results which have one of the LOB types can be created with the AS LOCATOR modifier. In this case, the entire LOB value is not materialized prior to invocation. Instead, a LOB LOCATOR is passed to the UDF. You can also use this capability on UDF parameters or results which have a distinct type that is based on a LOB. This capability is limited to non-SQL UDFs. Note that the argument to such a function can be any LOB value of the defined type; it does not have to be a host variable defined as one of the LOCATOR types. The use of host variable locators as arguments is completely unrelated to the use of AS LOCATOR in UDF parameters and result definitions.
- UDFs can be defined with distinct types as parameters or as the result. (Earlier examples have illustrated this.) DB2 will pass the value to the UDF in the format of the source data type of the distinct type.

Distinct type values that originate in a host variable and which are used as arguments to a UDF which has its corresponding parameter defined as a distinct type, **must be explicitly cast to the distinct type by the user**. There is no host language type for distinct types. DB2's strong typing necessitates this. Otherwise your results may be ambiguous. So, consider the BOAT distinct type that is defined over a BLOB, and consider the BOAT_COST UDF from "Example: External function with UDT parameter" on page 208, that takes an object of type BOAT as its argument. In the following fragment of a C language application, the host variable :ship holds the BLOB value that is to be passed to the BOAT_COST function:

```
EXEC SQL BEGIN DECLARE SECTION;
      SQL TYPE IS BLOB(150K) ship;
EXEC SQL END DECLARE SECTION;
```

Both of the following statements correctly resolve to the BOAT_COST function, because both cast the :ship host variable to type BOAT:

```
... SELECT BOAT_COST (BOAT(:ship)) FROM ...
... SELECT BOAT_COST (CAST(:ship AS BOAT)) FROM ...
```

If there are multiple BOAT distinct types in the database, or BOAT UDFs in other schema, you must exercise care with your function path. Otherwise your results may be ambiguous.

User-defined distinct types (UDT)

A user-defined distinct type is a mechanism that allows you to extend DB2 capabilities beyond the built-in data types available. User-defined distinct types enable you to define new data types to DB2 which gives you considerable power since you are no longer restricted to using the system-supplied built-in data types to model your business and capture the semantics of your data. Distinct data types allow you to map on a one-to-one basis to existing database types.

The following topics describe UDTs in more detail:

- "Why use UDTs?"
- "Defining a UDT" on page 216
- "Defining tables with UDTs" on page 217
- "Manipulating UDTs" on page 218
- "Synergy between UDTs, UDFs, and LOBs" on page 222

Why use UDTs?

There are several benefits associated with UDTs:

1. Extensibility.

By defining new types, you can indefinitely increase the set of types provided by DB2 to support your applications.

2. Flexibility.

You can specify any semantics and behavior for your new type by using user-defined functions (UDFs) to augment the diversity of the types available in the system.

3. Consistent behavior.

Strong typing insures that your UDTs will behave appropriately. It guarantees that only functions defined on your UDT can be applied to instances of the UDT.

4. Encapsulation.

The behavior of your UDTs is restricted by the functions and operators that can be applied on them. This provides flexibility in the implementation since running applications do not depend on the internal representation that you chose for your type.

5. **Extensible behavior.**

The definition of user-defined functions on types can augment the functionality provided to manipulate your UDT at any time. (See “User-defined functions (UDF)” on page 200)

6. **Foundation for object-oriented extensions.**

UDTs are the foundation for most object-oriented features. They represent the most important step towards object-oriented extensions.

Defining a UDT

UDTs, like other objects such as tables, indexes, and UDFs, need to be defined with a CREATE statement.

Use the CREATE DISTINCT TYPE statement to define your new UDT. See CREATE DISTINCT TYPE in the SQL Reference for detailed explanations of the statement syntax and all its options.

For the CREATE DISTINCT TYPE statement, note that:

1. The name of the new UDT can be a qualified or an unqualified name.
2. The source type of the UDT is the type used by DB2 to internally represent the UDT. For this reason, it must be a built-in data type. Previously defined UDTs cannot be used as source types of other UDTs.

As part of a UDT definition, DB2 always generates cast functions to:

- Cast from the UDT to the source type, using the standard name of the source type. For example, if you create a distinct type based on FLOAT, the cast function called DOUBLE is created.
- Cast from the source type to the UDT. See the SQL Reference for a discussion of when additional casts to the UDTs are generated.

These functions are important for the manipulation of UDTs in queries.

Resolving unqualified UDTs

The function path is used to resolve any references to an unqualified type name or function, except if the type name or function is

- Created
- Dropped
- Commented on.

For information on how unqualified function references are resolved, see “Using qualified function reference” on page 212.

Examples: Using CREATE DISTINCT TYPE

The following are examples of using CREATE DISTINCT TYPE:

- Example: Money
- Example: Resume

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Example: Money

Suppose you are writing applications that need to handle different currencies and wish to ensure that DB2 does not allow these currencies to be compared or manipulated directly with one another in queries. Remember that conversions are necessary whenever you want to compare values of different currencies. So you define as many UDTs as you need; one for each currency that you may need to represent:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,2)
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,2)
CREATE DISTINCT TYPE GERMAN_MARK AS DECIMAL (9,2)
```

Example: Resume

Suppose you would like to keep the application forms that are filled out by applicants to your company in a DB2 table and you are going to use functions to extract the information from these forms. Because these functions cannot be applied to regular character strings (because they are certainly not able to find the information they are supposed to return), you define a UDT to represent the filled forms:

```
CREATE DISTINCT TYPE PERSONAL.APPLICATION_FORM AS CLOB(32K)
```

Defining tables with UDTs

After you have defined several UDTs, you can start defining tables with columns whose types are UDTs. Following are examples using CREATE TABLE:

- Example: Sales
- Example: Application forms

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Example: Sales

Suppose you want to define tables to keep your company’s sales in different countries as follows:

```
CREATE TABLE US_SALES
(PRODUCT_ITEM INTEGER,
 MONTH        INTEGER CHECK (MONTH BETWEEN 1 AND 12),
 YEAR         INTEGER CHECK (YEAR > 1985),
 TOTAL        US_DOLLAR)
```

```
CREATE TABLE CANADIAN_SALES
(PRODUCT_ITEM INTEGER,
 MONTH        INTEGER CHECK (MONTH BETWEEN 1 AND 12),
 YEAR         INTEGER CHECK (YEAR > 1985),
 TOTAL        CANADIAN_DOLLAR)
```

```
CREATE TABLE GERMAN_SALES
(PRODUCT_ITEM INTEGER,
 MONTH        INTEGER CHECK (MONTH BETWEEN 1 AND 12),
 YEAR         INTEGER CHECK (YEAR > 1985),
 TOTAL        GERMAN_MARK)
```

The UDTs in the above examples are created using the same CREATE DISTINCT TYPE statements in “Example: Money”. Note that the above examples use check constraints. For information about check constraints see the “Adding and using check constraints” on page 135.

Example: Application forms

Suppose you need to define a table in order to keep the forms filled out by applicants as follows:

```

CREATE TABLE APPLICATIONS
  (ID          INTEGER,
   NAME       VARCHAR (30),
   APPLICATION_DATE DATE,
   FORM       PERSONAL.APPLICATION_FORM)

```

You have fully qualified the UDT name because its qualifier is not the same as your authorization ID and you have not changed the default function path. Remember that whenever type and function names are not fully qualified, DB2 searches through the schemas listed in the current function path and looks for a type or function name matching the given unqualified name.

Manipulating UDTs

One of the most important concepts associated with UDTs is *strong typing*. Strong typing guarantees that only functions and operators defined on the UDT can be applied to its instances.

Strong typing is important to ensure that the instances of your UDTs are correct. For example, if you have defined a function to convert US dollars to Canadian dollars according to the current exchange rate, you do not want this same function to be used to convert German marks to Canadian dollars because it will certainly return the wrong amount.

As a consequence of strong typing, DB2 does not allow you to write queries that compare, for example, UDT instances with instances of the UDT source type. For the same reason, DB2 will not let you apply functions defined on other types to UDTs. If you want to compare instances of UDTs with instances of another type, you have to cast the instances of one or the other type. In the same sense, you have to cast the UDT instance to the type of the parameter of a function that is not defined on a UDT if you want to apply this function to a UDT instance.

For an example of manipulating UDTs, see “Examples of manipulating UDTs”.

Examples of manipulating UDTs

The following are examples of manipulating UDTs:

- Example: Comparisons between UDTs and constants
- Example: Casting between UDTs
- Example: Comparisons involving UDTs
- Example: Sourced UDFs involving UDTs
- Example: Assignments involving UDTs
- Example: Assignments in dynamic SQL
- Example: Assignments involving different UDTs
- Example: Use of UDTs in UNION

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Example: Comparisons between UDTs and constants

Suppose you want to know which products sold more than US \$100 000.00 in the US in the month of July, 1992 (7/92).

```

SELECT PRODUCT_ITEM
FROM   US_SALES
WHERE  TOTAL > US_DOLLAR (100000)
AND    month = 7
AND    year  = 1992

```


Because you cannot compare US dollars with instances of the source type of US dollars (that is, DECIMAL) directly, you have used the cast function provided by DB2 to cast from DECIMAL to US dollars. You can also use the other cast function provided by DB2 (that is, the one to cast from US dollars to DECIMAL) and cast the column total to DECIMAL. Either way you decide to cast, from or to the UDT, you can use the cast specification notation to perform the casting, or the functional notation. That is, you could have written the above query as:

```

SELECT PRODUCT_ITEM
FROM   US_SALES
WHERE  TOTAL > CAST (100000 AS us_dollar)
AND    MONTH = 7
AND    YEAR  = 1992

```

Example: Casting between UDTs

Suppose you want to define a UDF that converts Canadian dollars to U.S. dollars. Suppose you can obtain the current exchange rate from a file managed outside of DB2. You would then define a UDF that obtains a value in Canadian dollars, accesses the exchange rate file and returns the corresponding amount in U.S. dollars.

At first glance, such a UDF may appear easy to write. However, not all C compilers support DECIMAL values. The UDTs representing different currencies have been defined as DECIMAL. Your UDF will need to receive and return DOUBLE values, since this is the only data type provided by C that allows the representation of a DECIMAL value without losing the decimal precision. Thus, your UDF should be defined as follows:

```

CREATE FUNCTION CDN_TO_US_DOUBLE(DOUBLE) RETURNS DOUBLE
EXTERNAL NAME 'MYLIB/CURRENCIES(C_CDN_US)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
NOT DETERMINISTIC

```

The exchange rate between Canadian and U.S. dollars may change between two invocations of the UDF, so you declare it as NOT DETERMINISTIC.

The question now is, how do you pass Canadian dollars to this UDF and get U.S. dollars from it? The Canadian dollars must be cast to DECIMAL values. The DECIMAL values must be cast to DOUBLE. You also need to have the returned DOUBLE value cast to DECIMAL and the DECIMAL value cast to U.S. dollars.

Such casts are performed automatically by DB2 anytime you define sourced UDFs, whose parameter and return type do not exactly match the parameter and return type of the source function. Therefore, you need to define two sourced UDFs. The first brings the DOUBLE values to a DECIMAL representation. The second brings the DECIMAL values to the UDT. Define the following:

```

CREATE FUNCTION CDN_TO_US_DEC (DECIMAL(9,2)) RETURNS DECIMAL(9,2)
SOURCE CDN_TO_US_DOUBLE (DOUBLE)

CREATE FUNCTION US_DOLLAR (CANADIAN_DOLLAR) RETURNS US_DOLLAR
SOURCE CDN_TO_US_DEC (DECIMAL())

```

Note that an invocation of the US_DOLLAR function as in US_DOLLAR(C1), where C1 is a column whose type is Canadian dollars, has the same effect as invoking:

```

US_DOLLAR (DECIMAL(CDN_TO_US_DOUBLE (DOUBLE (DECIMAL (C1))))))

```

That is, C1 (in Canadian dollars) is cast to decimal which in turn is cast to a double value that is passed to the CDN_TO_US_DOUBLE function. This function

accesses the exchange rate file and returns a double value (representing the amount in U.S. dollars) that is cast to decimal, and then to U.S. dollars.

A function to convert German marks to U.S. dollars would be similar to the example above:

```
CREATE FUNCTION GERMAN_TO_US_DOUBLE(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/CURRENCIES(C_GER_US)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC

CREATE FUNCTION GERMAN_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  SOURCE GERMAN_TO_US_DOUBLE(DOUBLE)

CREATE FUNCTION US_DOLLAR(GERMAN_MARK) RETURNS US_DOLLAR
  SOURCE GERMAN_TO_US_DEC (DECIMAL())
```

Example: Comparisons involving UDTs

Suppose you want to know which products sold more in the US than in Canada and Germany for the month of March, 1989 (3/89):

```
SELECT US.PRODUCT_ITEM, US.TOTAL
  FROM US_SALES AS US, CANADIAN_SALES AS CDN, GERMAN_SALES AS GERMAN
 WHERE US.PRODUCT_ITEM = CDN.PRODUCT_ITEM
 AND US.PRODUCT_ITEM = GERMAN.PRODUCT_ITEM
 AND US.TOTAL > US_DOLLAR (CDN.TOTAL)
 AND US.TOTAL > US_DOLLAR (GERMAN.TOTAL)
 AND US.MONTH = 3
 AND US.YEAR = 1989
 AND CDN.MONTH = 3
 AND CDN.YEAR = 1989
 AND GERMAN.MONTH = 3
 AND GERMAN.YEAR = 1989
```

Because you cannot directly compare US dollars with Canadian dollars or German Marks, you use the UDF to cast the amount in Canadian dollars to US dollars, and the UDF to cast the amount in German Marks to US dollars. You cannot cast them all to DECIMAL and compare the converted DECIMAL values because the amounts are not monetarily comparable as they are not in the same currency.

Example: Sourced UDFs involving UDTs

Suppose you have defined a sourced UDF on the built-in SUM function to support SUM on German Marks:

```
CREATE FUNCTION SUM (GERMAN_MARKS)
  RETURNS GERMAN_MARKS
  SOURCE SYSIBM.SUM (DECIMAL())
```

You want to know the total of sales in Germany for each product in the year of 1994. You would like to obtain the total sales in US dollars:

```
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
 WHERE YEAR = 1994
 GROUP BY PRODUCT_ITEM
```

You could not write `SUM (us_dollar (total))`, unless you had defined a SUM function on US dollar in a manner similar to the above.

Example: Assignments involving UDTs

Suppose you want to store the form filled out by a new applicant into the database. You have defined a host variable containing the character string value used to represent the filled form:

```
EXEC SQL BEGIN DECLARE SECTION;
      SQL TYPE IS CLOB(32K) hv_form;
EXEC SQL END DECLARE SECTION;

/* Code to fill hv_form */

INSERT INTO APPLICATIONS
      VALUES (134523, 'Peter Holland', CURRENT DATE, :hv_form)
```

You do not explicitly invoke the cast function to convert the character string to the UDT `personal.application_form`. This is because DB2 allows you to assign instances of the source type of a UDT to targets having that UDT.

Example: Assignments in dynamic SQL

If you want to use the same statement given in “Example: Assignments involving UDTs” in dynamic SQL, you can use parameter markers as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
      long id;
      char name[30];
      SQL TYPE IS CLOB(32K) form;
      char command[80];
EXEC SQL END DECLARE SECTION;

/* Code to fill host variables */

strcpy(command, "INSERT INTO APPLICATIONS VALUES");
strcat(command, "(?, ?, CURRENT DATE, ?)");

EXEC SQL PREPARE APP_INSERT FROM :command;
EXEC SQL EXECUTE APP_INSERT USING :id, :name, :form;
```

You made use of DB2’s cast specification to tell DB2 that the type of the parameter marker is `CLOB(32K)`, a type that is assignable to the UDT column. Remember that you cannot declare a host variable of a UDT type, since host languages do not support UDTs. Therefore, you cannot specify that the type of a parameter marker is a UDT.

Example: Assignments involving different UDTs

Suppose you have defined two sourced UDFs on the built-in `SUM` function to support `SUM` on US and Canadian dollars, similar to the UDF sourced on German Marks in “Example: Sourced UDFs involving UDTs” on page 220:

```
CREATE FUNCTION SUM (CANADIAN_DOLLAR)
      RETURNS CANADIAN_DOLLAR
      SOURCE SYSIBM.SUM (DECIMAL())

CREATE FUNCTION SUM (US_DOLLAR)
      RETURNS US_DOLLAR
      SOURCE SYSIBM.SUM (DECIMAL())
```

Now suppose your supervisor requests that you maintain the annual total sales in US dollars of each product and in each country, in separate tables:

```
CREATE TABLE US_SALES_94
      (PRODUCT_ITEM INTEGER,
      TOTAL US_DOLLAR)
```

```

CREATE TABLE GERMAN_SALES_94
(PRODUCT_ITEM INTEGER,
TOTAL US_DOLLAR)

CREATE TABLE CANADIAN_SALES_94
(PRODUCT_ITEM INTEGER,
TOTAL US_DOLLAR)

INSERT INTO US_SALES_94
SELECT PRODUCT_ITEM, SUM (TOTAL)
FROM US_SALES
WHERE YEAR = 1994
GROUP BY PRODUCT_ITEM

INSERT INTO GERMAN_SALES_94
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
FROM GERMAN_SALES
WHERE YEAR = 1994
GROUP BY PRODUCT_ITEM

INSERT INTO CANADIAN_SALES_94
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
FROM CANADIAN_SALES
WHERE YEAR = 1994
GROUP BY PRODUCT_ITEM

```

You explicitly cast the amounts in Canadian dollars and German Marks to US dollars since different UDTs are not directly assignable to each other. You cannot use the cast specification syntax because UDTs can only be cast to their own source type.

Example: Use of UDTs in UNION

Suppose you would like to provide your American users with a query to show all the sales of every product of your company:

```

SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
FROM US_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
FROM CANADIAN_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
FROM GERMAN_SALES

```

You cast Canadian dollars to US dollars and German Marks to US dollars because UDTs are union compatible only with the same UDT. You must use the functional notation to cast between UDTs since the cast specification only allows you to cast between UDTs and their source types.

Synergy between UDTs, UDFs, and LOBs

In previous sections, you learned how to define and use the individual DB2 object extensions (UDTs, UDFs, and LOBs). However, as you will see in this section, there is a lot of synergy between these three object extensions.

See the following sections for more details,

- “Combining UDTs, UDFs, and LOBs” on page 223
- “Examples of complex applications” on page 223

Combining UDTs, UDFs, and LOBs

According to the concept of object-orientation, similar objects in the application domain are grouped into related types. Each of these types have a name, an internal representation, and behavior. By using UDTs, you can tell DB2 the name of your new type and how it is internally represented. A LOB is one of the possible internal representations for your new type and is the most suitable representation for large, complex structures. By using UDFs, you can define the behavior of the new type. Consequently, there is an important synergy between UDTs, UDFs, and LOBs. An application type with a complex data structure and behavior is modeled as a UDT that is internally represented as a LOB, with its behavior implemented by UDFs. The rules governing the semantic integrity of your application type will be represented as constraints and triggers. To have better control and organization of your related UDTs and UDFs, you should keep them in the same schema.

Examples of complex applications

The following examples show how you can use UDTs, UDFs, and LOBs together in complex applications:

Example: Defining the UDT and UDFs

Example: Using LOB function to populate the database

Example: Using UDFs to query instances of UDTs

Example: Using LOB locators to manipulate UDT instances

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Example: Defining the UDT and UDFs

Suppose you would like to keep the electronic mail (e-mail) sent to your company in DB2 tables. Ignoring any issues of privacy, you plan to write queries over such e-mail to find out their subject, how often your e-mail service is used to receive customer orders, and so on. E-mail can be quite large, and it has a complex internal structure (a sender, a receiver, the subject, date, and the e-mail content). Therefore, you decide to represent the e-mail by means of a UDT whose source type is a large object. You define a set of UDFs on your e-mail type, such as functions to extract the subject of the e-mail, the sender, the date, and so on. You also define functions that can perform searches on the content of the e-mail. You do the above using the following CREATE statements:

```
CREATE DISTINCT TYPE E_MAIL AS BLOB (1M)
```

```
CREATE FUNCTION SUBJECT (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(SUBJECT)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
```

```
CREATE FUNCTION SENDER (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(SENDER)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
```

```
CREATE FUNCTION RECEIVER (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(RECEIVER)'
```

```

LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION

CREATE FUNCTION SENDING_DATE (E_MAIL)
RETURNS DATE CAST FROM VARCHAR(10)
EXTERNAL NAME 'LIB/PGM(SENDING_DATE)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION

CREATE FUNCTION CONTENTS (E_MAIL)
RETURNS BLOB (1M)
EXTERNAL NAME 'LIB/PGM(CONTENTS)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION

CREATE FUNCTION CONTAINS (E_MAIL, VARCHAR (200))
RETURNS INTEGER
EXTERNAL NAME 'LIB/PGM(CONTAINS)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION

CREATE TABLE ELECTRONIC_MAIL
  (ARRIVAL_TIMESTAMP TIMESTAMP,
  MESSAGE E_MAIL)

```

Example: Using LOB function to populate the database

Suppose you populate your table by transferring your e-mail that is maintained in files into DB2. You would execute the following INSERT statement multiple times with different values of the HV_EMAIL_FILE until you have stored all your e_mail into DB2:

```

EXEC SQL BEGIN DECLARE SECTION
  SQL TYPE IS BLOB_FILE HV_EMAIL_FILE;

EXEC SQL END DECLARE SECTION
  strcpy (HV_EMAIL_FILE.NAME, "/u/mail/email/mbox");
  HV_EMAIL_FILE.NAME_LENGTH = strlen(HV_EMAIL_FILE.NAME);
  HV_EMAIL_FILE.FILE_OPTIONS = 2;

EXEC SQL INSERT INTO ELECTRONIC_MAIL
  VALUES (CURRENT_TIMESTAMP, :hv_email_file);

```

All the function provided by DB2 LOB support is applicable to UDTs whose source type are LOBs. Therefore, you have used LOB file reference variables to assign the contents of the file into the UDT column. You have not used the cast function to convert values of BLOB type into your e-mail type. This is because DB2 let you assign values of the source type of a distinct type to targets of the distinct type.

Example: Using UDFs to query instances of UDTs

Suppose you need to know how much e-mail was sent by a specific customer regarding customer orders and you have the e-mail address of your customers in the customers table.

```

SELECT COUNT (*)
FROM ELECTRONIC_MAIL AS EMAIL, CUSTOMERS
WHERE SUBJECT (EMAIL.MESSAGE) = 'customer order'
AND CUSTOMERS.EMAIL_ADDRESS = SENDER (EMAIL.MESSAGE)
AND CUSTOMERS.NAME = 'Customer X'

```

You have used the UDFs defined on the UDT in this SQL query since they are the only means to manipulate the UDT. In this sense, your UDT e-mail is completely encapsulated. Its internal representation and structure are hidden and can only be manipulated by the defined UDFs. These UDFs know how to interpret the data without the need to expose its representation.

Suppose you need to know the details of all the e-mail your company received in 1994 that had to do with the performance of your products in the marketplace.

```

SELECT SENDER (MESSAGE), SENDING_DATE (MESSAGE), SUBJECT (MESSAGE)
FROM ELECTRONIC_MAIL
WHERE CONTAINS (MESSAGE,
"performance" AND "products" AND "marketplace") = 1

```

You have used the contains UDF which is capable of analyzing the contents of the message searching for relevant keywords or synonyms.

Example: Using LOB locators to manipulate UDT instances

Suppose you would like to obtain information about a specific e-mail without having to transfer the entire e-mail into a host variable in your application program. (Remember that an e-mail can be quite large.) Since your UDT is defined on a LOB, you can use LOB locators for that purpose:

```

EXEC SQL BEGIN DECLARE SECTION
    long hv_len;
    char hv_subject[200];
    char hv_sender[200];
    char hv_buf[4096];
    char hv_current_time[26];
    SQL TYPE IS BLOB_LOCATOR hv_email_locator;
EXEC SQL END DECLARE SECTION

EXEC SQL SELECT MESSAGE
    INTO :hv_email_locator
    FROM ELECTRONIC_MAIL
    WHERE ARRIVAL_TIMESTAMP = :hv_current_time;

EXEC SQL VALUES (SUBJECT (E_MAIL(:hv_email_locator))
    INTO :hv_subject;
... code that checks if the subject of the e_mail is relevant ....
... if the e_mail is relevant, then.....

EXEC SQL VALUES (SENDER (CAST (:hv_email_locator AS E_MAIL)))
    INTO :hv_sender;

```

Because your host variable is of type BLOB locator (the source type of the UDT), you have explicitly converted the BLOB locator to your UDT, whenever it was used as an argument of a UDF defined on the UDT.

Using DataLinks

The DataLink data type is one of the basic building blocks for extending the types of data that can be stored in database files. The idea of a DataLink is that the actual data stored in the column is only a pointer to the object. This object can be anything, an image file, a voice recording, a text file, etc. The method used for resolving to the object is to store a Uniform Resource Locator (URL). This means

that a row in a table can be used to contain information about the object in traditional data types, and the object itself can be referenced using the DataLink data type. The user can use new SQL scalar functions to get back the path to the object and the server on which the object is stored. With the DataLink data type, there is a fairly loose relationship between the row and the object. For instance, deleting a row will sever the relationship to the object referenced by the DataLink, but the object itself might not be deleted.

An SQL table created with a DataLink column can be used to hold information about an object, without actually containing the object itself. This concept gives the user much more flexibility in the types of data that can be managed using an SQL table. If, for instance, the user has thousands of video clips stored in the integrated file system of their server, they may want to use an SQL table to contain information about these video clips. But since the user already has the objects stored in a directory, they simply want the SQL table to contain references to the objects, not contain the actual bytes of storage. A good solution would be to use DataLinks. The SQL table would use traditional SQL data types to contain information about each clip, such as title, length, date, etc. But the clip itself would be referenced using a DataLink column. Each row in the table would store a URL for the object and an optional comment. Then an application that is working with the clips can retrieve the URL using SQL interfaces, and then use a browser or other playback software to work with the URL and display the video clip.

There are several advantages to using this technique:

- The integrated file system can store any type of stream file.
- The integrated file system can store extremely large objects, that would not fit into a character column, or perhaps even a LOB column.
- The hierarchical nature of the integrated file system is well-suited to organizing and working with the stream file objects.
- By leaving the bytes of the object outside the database and in the integrated file system, applications can achieve better performance by allowing the SQL runtime engine to handle queries and reports, and allowing the file system to handle streaming of video, displaying images, text, etc.

Using DataLinks also gives control over the objects while they are in "linked" status. A DataLink column can be created such that the referenced object cannot be deleted, moved, or renamed while there is a row in the SQL table that references that object. This object would be considered linked. Once the row containing that reference is deleted, the object is unlinked. To understand this concept fully, one should know the levels of control that can be specified when creating a DataLink column. Refer to the SQL Reference for the exact syntax used when creating DataLink columns.

For more details on DataLinks, see the following sections:

- "NO LINK CONTROL" on page 227
- "FILE LINK CONTROL (with File System Permissions)" on page 227
- "FILE LINK CONTROL (with Database Permissions)" on page 227
- "Commands used for working with DataLinks" on page 227

NO LINK CONTROL

When a column is created with NO LINK CONTROL, there is no linking that takes place when rows are added to the SQL table. The URL is verified to be syntactically correct, but there is no check to make sure that the server is accessible, or that the file even exists.

FILE LINK CONTROL (with File System Permissions)

When the DataLink column is created as FILE LINK CONTROL with file system (FS) permissions, the system will verify that any DataLink value is a valid URL, with a valid server name and file name. The file must exist at the time that row is being inserted into the SQL table. When the object is found, it will be marked as linked. This means that the object cannot be moved, deleted, or renamed during the time that it is linked. Also, an object cannot be linked more than once. If the server name portion of the URL specifies a remote system, that system must be accessible. If a row containing a DataLink value is deleted, the object is unlinked. If a DataLink value is updated to a different value, the old object is unlinked, and the new object is linked.

The integrated file system is still responsible for managing permissions for the linked object. The permissions are not modified during the link or unlink processes. This option provides control of the object's existence for the duration of time that it is linked.

FILE LINK CONTROL (with Database Permissions)

When the DataLink column is create as FILE LINK CONTROL with database permissions, the URL is verified, and all existing permissions to the object are removed. The ownership of the object is changed to a special system-supplied user profile. During the time that the object is linked, the only access to the object is by obtaining the URL from the SQL table that has the object linked. This is handled by using a special access token that is appended to the URL returned by SQL. Without the access token, all attempts to access the object will fail with an authority violation. If the URL with the access token is retrieved from the SQL table by normal means (FETCH, SELECT INTO, etc.) the file system filter will validate the access token and allow the access to the object.

This option provides the control of preventing updates to the linked object for users trying to access the object by direct means. Since the only access to the object is by obtaining the access token from an SQL operation, an administrator can effectively control access to the linked objects by using the database permissions to the SQL table that contains the DataLink column.

Commands used for working with DataLinks

Support for the DataLink data type can be broken down into 3 different components:

1. The DB2 database support has a data type called DATALINK. This can be specified on SQL statements such as CREATE TABLE and ALTER TABLE. The column cannot have any default other than NULL. Access to the data must be using SQL interfaces. This is because the DATALINK itself is not compatible with any host variable type. SQL scalar functions can be used to retrieve the DATALINK value in character form. There is a DLVALUE scalar function that must be used in SQL to INSERT and UPDATE the values in the column.
2. The DataLink File Manager (DLFM) is the component that maintains the link status for the files on a server, and keeps track of meta-data for each file. This

code handles linking, unlinking, and commitment control issues. An important aspect of DataLinks is that the DLFM need not be on the same physical system as the SQL table that contains the DataLink column. So an SQL table can link an object that resides in either the same system's integrated file system, or a remote server's integrated file system.

3. The DataLink filter must be executed when the file system tries operations against files that are in directories designated as containing linked objects. This component determines if the file is linked, and optionally, if the user is authorized to access the file. If the file name includes an access token, the token will be verified. Since there is extra overhead in this filter process, it is only executed when the accessed object exists in one of the directories within a DataLink 'prefix'. See the discussion below on prefixes.

When working with DataLinks, there are several steps that must be taken to properly configure the system:

- TCP/IP must be configured on any systems that are going to be used when working with DataLinks. This would include the systems on which the SQL tables with DataLink columns are going to be created, as well as the systems that will contain the objects to be linked. In most cases, this will be the same system. Since the URL that is used to reference the object contains a TCP/IP server name, this name must be recognized by the system that is going to contain the DataLink. The command CFGTCP can be used to configure the TCP/IP names, or to register a TCP/IP name server.
- The system that contains the SQL tables must have the Relational Database Directory updated to reflect the local database system, and any optional remote systems. The command WRKRDBDIRE can be used to add or modify information in this directory. For consistency, it is recommended that the same names be used as the TCP/IP server name and the Relational Database name.
- The DLFM server must be started on any systems that will contain objects to be linked. The command STRTCPSVR *DLFM can be used to start the DLFM server. The DLFM server can be ended by using the CL command ENDTCPMSVR *DLFM.

Once the DLFM has been started, there are some steps needed to configure the DLFM. These DLFM functions are available via an executable script that can be entered from the QShell interface. To get to the interactive shell interface, use the CL command QSH. This will bring up a command entry screen from which you can enter the DLFM script commands. The script command dfmadmin -help can be used to display help text and syntax diagrams. For the most commonly used functions, CL commands have also been provided. Using the CL commands, most or all of the DLFM configuration can be accomplished without using the script interface. Depending on your preferences, you can choose to use either the script commands from the QSH command entry screen or the CL commands from the CL command entry screen.

Since these functions are meant for a system administrator or a database administrator, they all require the *IOSYSCFG special authority.

Adding a prefix - A prefix is a path or directory that will contain objects to be linked. When setting up the DLFM on a system, the administrator must add any prefixes that will be used for DataLinks. The script command dfmadmin -add_prefix is used to add prefixes. The CL command to add prefixes is ADDPFXDLFM.

For instance, on server TESTSYS1, there is a directory called /mydir/datalinks/ that contains the objects that will be linked. The administrator uses the command `ADDPFXDLFM PREFIX('/mydir/datalinks/')` to add the prefix. Now links for URLs such as:

```
http://TESTSYS1/mydir/datalinks/videos/file1.mpg
```

or

```
file://TESTSYS1/mydir/datalinks/text/story1.txt
```

would be valid since their path begins with a valid prefix.

It is also possible to remove a prefix using the script command `dfmadmin -del_prefix`. This is not a commonly used function since it can only be executed if there are no linked objects anywhere in the directory structure contained within the prefix name.

Adding a Host Database - A host database is a relational database system from which a link request originates. If the DLFM is on the same system as the SQL tables that will contain the DataLinks, then only the local database name needs to be added. If the DLFM will have link requests coming from remote systems, then all of their names must be registered with the DLFM. The script command to add a host database is `dfmadmin -add_db` and the CL command is `ADDHDBDLFM`. This function also requires that the libraries containing the SQL tables also be registered.

For instance, on server TESTSYS1, where you have already added the /mydir/datalinks/ prefix, you want SQL tables on the local system in either library TESTDB or PRODDB to be allowed to link objects on this server. Use the following command: **ADDHDBDLFM HOSTDBLIB((TESTDB) (PRODDB)) HOSTDB(TESTSYS1)**

Once the DLFM has been started, and the prefixes and host database names have been registered, you can begin linking objects in the file system.

Chapter 13. Writing User-Defined Functions (UDFs)

User Defined Functions (UDFs) consist of three types: sourced, external, and SQL. Sourced function UDFs call other functions to perform the operation. SQL and external function UDFs require that you write and execute separate code. This chapter is about writing SQL and external functions. To write external and SQL functions, you need to do the following:

- Understand the “UDF runtime environment”
- Register the UDF, so that it is known to the database
- Write the function code to perform the function and pass the appropriate parameters
- Debug and test the function.

For examples of writing functions, see

- “Example: Square of a number UDF” on page 242
- “Example: Counter” on page 244

UDF runtime environment

There are several things to consider about the environment in which a UDF executes and the limitations of that environment. These factors should be considered carefully if you are contemplating writing complex function code for UDFs.

The factors are:

- “Length of time that the UDF runs”
- “Threads considerations” on page 232
- “Parallel processing” on page 232

Length of time that the UDF runs

UDFs are invoked from within an SQL statement execution, which is normally a query operation that potentially runs against thousands of rows in a table. Because of this, the UDF needs to be invoked from a low level of the database.

As a consequence of being invoked from such a low level, there are certain resources (locks and seizures) being held at the time the UDF is invoked and for the duration of the UDF execution. These resources are primarily locks on any tables and indexes involved in the SQL statement that is invoking the UDF. Due to these held resources, it is important that the UDF not perform operations that may take an extended period of time (minutes or hours). Because of the critical nature of holding resources for long periods of time, the database only waits for a certain period of time for the UDF to finish. If the UDF does not finish in the time allocated, the SQL statement will fail, which can be quite aggravating to the end user.

The default UDF wait time used by the database should be more than sufficient to allow a normal UDF to run to completion. However, if you have a long running UDF and wish to increase the wait time, this can be done using the `UDF_TIME_OUT` option in the query INI file. See Query Options File `QAQQINI` in the *Database Performance and Query Optimization* information for details on the INI

file. Note, however, that there is a maximum time limit that the database will not exceed, regardless of the value specified for UDF_TIME_OUT.

Since resources are held while the UDF is run, it is important that the UDF not operate on the same tables or indexes allocated for the original SQL statement or, if it does, that it does not perform an operation that conflicts with the one being performed in the SQL statement. Specifically, the UDF should not try to perform any insert, update or delete row operation on those tables.

Threads considerations

A UDF, defined as FENCED, runs in the same job as the SQL statement that invoked it. However, the UDF runs in a system thread, separate from the thread that is running the SQL statement. For more information about threads, see Database considerations for multithreaded programming in the Programming category of the Information Center.

Because the UDF runs in the same job as the SQL statement, it shares much of the same environment as the SQL statement. However, because it runs under a separate thread, the following threads considerations apply:

- The UDF will conflict with thread level resources held by the SQL statement's thread. Primarily, these are the table resources discussed above.
- UDFs do not inherit any program adopted authority that may have been active at the time the SQL statement was invoked. UDF authority comes from either the authority associated with the UDF program itself or the authority of the user running the SQL statement.
- The UDF cannot perform any operation that is blocked from being run in a secondary thread.
- The UDF program must be created such that it either runs under a named activation group or in the activation group of its caller (ACTGRP parameter). Programs that specify ACTGRP(*NEW) will not be allowed to run as UDFs.

For information about defining a function as UNFENCED, see "Making a function fenced or unfenced" on page 242.

Parallel processing

A UDF can be defined to allow parallel processing. This means that the same UDF program can be running in multiple threads at the same time. Therefore, if ALLOW PARALLEL is specified for the UDF, ensure that it is thread safe. For more information about threads, see Database considerations for multithreaded programming in the Programming category of the iSeries Information Center.

User-defined table functions cannot run in parallel; therefore, DISALLOW PARALLEL must be specified when creating the function

Writing function code

Writing function code involves knowing how to write the SQL or external function to perform the function. It also involves understanding the interface between the database and the function code to define it correctly, and determining packaging options when creating the executable program.

You can write UDFs as SQL functions or as external functions. For more details, see

- "Writing UDFs as SQL functions" on page 233

- “Writing UDFs as external functions” on page 234
- See also “Table function considerations” on page 240.

Writing UDFs as SQL functions

SQL functions are UDFs that you have defined, written, and registered using the CREATE FUNCTION statement. As such, they are written using only the SQL language and their definition is completely contained within one (potentially large) CREATE FUNCTION statement. The creation of an SQL function causes the registration of the UDF, generates the executable code for the function, and defines to the database the details of how parameters are actually passed. Therefore, writing these functions is quite clean and provides less chance of introducing errors into the function.

Scalar UDFs

The CREATE FUNCTION statement for SQL scalar functions follow the general flow of:

```
CREATE FUNCTION function-name(parameters) RETURNS return-value
LANGUAGE SQL
BEGIN
    sql-statements
END
```

For example, a function that returns a priority based on a date:

```
CREATE FUNCTION PRIORITY(indate DATE) RETURNS CHAR(7)
LANGUAGE SQL
BEGIN
RETURN(
    CASE WHEN indate>CURRENT DATE-3 DAYS THEN 'HIGH'
         WHEN indate>CURRENT DATE-7 DAYS THEN 'MEDIUM'
         ELSE 'LOW'
    END
);
END
```

The function could then be invoked as:

```
SELECT ORDERNBR, PRIORITY(ORDERDUEDATE) FROM ORDERS
```

Table UDFs

The CREATE FUNCTION statement for SQL table functions follow the general flow of:

```
CREATE FUNCTION function-name(parameters) RETURNS TABLE return-columns
LANGUAGE SQL
BEGIN
    sql-statements
RETURN
    select-statement
END
```

For example, a function that returns data based on a date:

```
CREATE FUNCTION PROJFUNC(indate DATE)
RETURNS TABLE (PROJNO CHAR(6), ACTNO SMALLINT, ACTSTAFF DECIMAL(5,2),
                ACSTDATE DATE, ACENDATE DATE)
LANGUAGE SQL
BEGIN
RETURN SELECT * FROM PROJECT
WHERE ACSTDATE<=indate;
END
```

The function could then be invoked as:

```
SELECT * FROM TABLE(PROJFUNC(:datehv)) X
```

SQL table functions are required to have one and only one RETURN statement.

Writing UDFs as external functions

You can also write the executable code of a UDF in a language other than SQL. While this method is slightly more cumbersome than an SQL function, it provides the flexibility for you to use whatever language is most effective for you. The executable code can be contained in either a program or service program.

External functions can also be written in Java. For a description of the parameters, see Java SQL Routines in the IBM Developer Kit for Java topic.

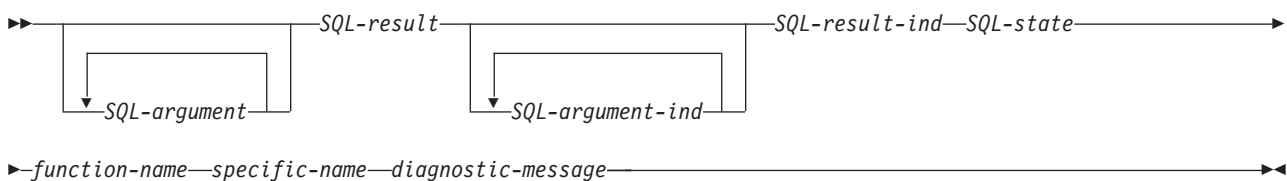
Passing arguments from DB2 to external functions

DB2 provides the storage for all parameters passed to a UDF. Therefore, parameters are passed to the external function by address. This is the normal parameter passing method for programs. For service programs, ensure that the parameters are defined correctly in the function code.

When defining and using the parameters in the UDF, care should be taken to ensure that no more storage is referenced for a given parameter than is defined for that parameter. The parameters are all stored in the same space and exceeding a given parameter's storage space can overwrite another parameter's value. This, in turn, can cause the function to see invalid input data or cause the value returned to the database to be invalid.

There are several supported parameter styles available to external UDFs. For the most part, the styles differ in how many parameters are passed to the external program or service program.

Parameter style SQL: The parameter style SQL conforms to the industry standard Structured Query Language (SQL). This parameter style can only be used with scalar UDFs. With parameter style SQL, the parameters are passed into the external program as follows (in the order specified):



SQL-argument

This argument is set by DB2 before calling the UDF. This value repeats n times, where n is the number of arguments specified in the function reference. The value of each of these arguments is taken from the expression specified in the function invocation. It is expressed in the data type of the defined parameter in the create function statement. Note: These parameters are treated as input only; any changes to the parameter values made by the UDF are ignored by DB2.

SQL-result

This argument is set by the UDF before returning to DB2. The database provides the storage for the return value. Since the parameter is passed by address, the address is of the storage where the return value should be placed. The database provides as much storage as needed for the return

value as defined on the CREATE FUNCTION statement. If the CAST FROM clause is used in the CREATE FUNCTION statement, DB2 assumes the UDF returns the value as defined in the CAST FROM clause, otherwise DB2 assumes the UDF returns the value as defined in the RETURNS clause.

SQL-argument-ind

This argument is set by DB2 before calling the UDF. It can be used by the UDF to determine if the corresponding *SQL-argument* is null or not. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument*, described previously. Each indicator is defined as a two-byte signed integer. It is set to one of the following values:

0 The argument is present and not null.

-1 The argument is null.

If the function is defined with RETURNS NULL ON NULL INPUT, the UDF does not need to check for a null value. However, if it is defined with CALLS ON NULL INPUT, any argument can be NULL and the UDF should check for null input. Note: these parameters are treated as input only; any changes to the parameter values made by the UDF are ignored by DB2.

SQL-result-ind

This argument is set by the UDF before returning to DB2. The database provides the storage for the return value. The argument is defined as a two-byte signed integer. If set to a negative value, the database interprets the result of the function as null. If set to zero or a positive value, the database uses the value returned in *SQL-result*. The database provides the storage for the return value indicator. Since the parameter is passed by address, the address is of the storage where the indicator value should be placed.

SQL-state

This argument is a CHAR(5) value that represents the SQLSTATE.

This parameter is passed in from the database set to '00000' and can be set by the function as a result state for the function. While normally the SQLSTATE is not set by the function, it can be used to signal an error or warning to the database as follows:

01Hxx The function code detected a warning situation. This results in an SQL warning. Here *xx* may be one of several possible strings.

38xxx The function code detected an error situation. It results in a SQL error. Here *xxx* may be one of several possible strings.

See SQL Messages and Codes for more information about valid SQLSTATES that the function may use.

function-name

This argument is set by DB2 before calling the UDF. It is a VARCHAR(139) value that contains the name of the function on whose behalf the function code is being invoked.

The form of the function name that is passed is:

<*schema-name*>.<*function-name*>

This parameter is useful when the function code is being used by multiple UDF definitions so that the code can distinguish which definition is being

invoked. Note: This parameter is treated as input only; any changes to the parameter value made by the UDF are ignored by DB2.

specific-name

This argument is set by DB2 before calling the UDF. It is a VARCHAR(128) value that contains the specific name of the function on whose behalf the function code is being invoked.

Like function-name, this parameter is useful when the function code is being used by multiple UDF definitions so that the code can distinguish which definition is being invoked. See the CREATE FUNCTION statement for more information about *specific-name*. Note: This parameter is treated as input only; any changes to the parameter value made by the UDF are ignored by DB2.

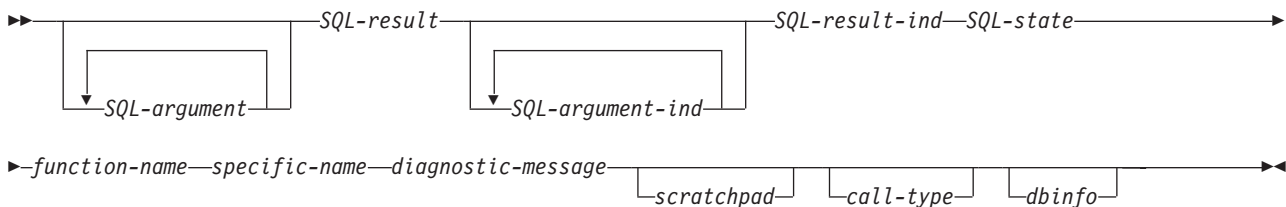
diagnostic-message

This argument is set by DB2 before calling the UDF. It is a VARCHAR(70) value that can be used by the UDF to send message text back when an SQLSTATE warning or error is signaled by the UDF.

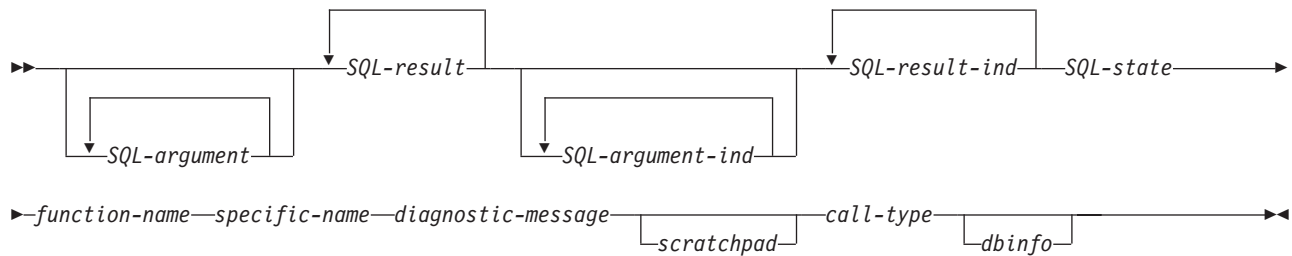
It is initialized by the database on input to the UDF and may be set by the UDF with descriptive information. Message text is ignored by DB2 unless the SQL-state parameter is set by the UDF.

| **Parameter style DB2SQL:** With the DB2SQL parameter style, the same parameters and same order of parameters are passed into the external program or service program as are passed in for parameter style SQL. However, DB2SQL allows additional optional parameters to be passed along as well. If more than one of the optional parameters below is specified in the UDF definition, they are passed to the UDF in the order defined below. Refer to parameter style SQL for the common parameters. This parameter style can be used for both scalar and table UDFs.

For scalar functions:



For table functions:



scratchpad

This argument is set by DB2 before calling the UDF. It is only present if the CREATE FUNCTION statement for the UDF specified the SCRATCHPAD keyword. This argument is a structure with the following elements:

- An INTEGER containing the length of the scratchpad.
- The actual scratchpad, initialized to all binary 0's by DB2 before the first call to the UDF.

The scratchpad can be used by the UDF either as working storage or as persistent storage, since it is maintained across UDF invocations.

For table functions, the scratchpad is initialized as above prior to the FIRST call to the UDF if FINAL CALL is specified on the CREATE FUNCTION. After this call, the scratchpad content is totally under control of the table function. DB2 does not examine or change the content of the scratchpad thereafter. The scratchpad is passed to the function on each invocation. The function can be re-entrant, and DB2 preserves its state information in the scratchpad.

If NO FINAL CALL was specified or defaulted for a table function, then the scratchpad is initialized as above for each OPEN call, and the scratchpad content is completely under control of the table function between OPEN calls. This can be very important for a table function used in a join or subquery. If it is necessary to maintain the content of the scratchpad across OPEN calls, then FINAL CALL must be specified in your CREATE FUNCTION statement. With FINAL CALL specified, in addition to the normal OPEN, FETCH, and CLOSE calls, the table function will also receive FIRST and FINAL calls, for the purpose of scratchpad maintenance and resource release.

call-type

This argument is set by DB2 before calling the UDF. For scalar functions, it is only present if the CREATE FUNCTION statement for the UDF specified the FINAL CALL keyword. However, for table functions it is *always* present. It follows the *scratchpad* argument; or the *diagnostic-message* argument if the scratchpad argument is not present. This argument takes the form of an INTEGER value.

For scalar functions:

- 1 This is the *first call* to the UDF for this statement. A first call is a *normal call* in that all SQL argument values are passed.

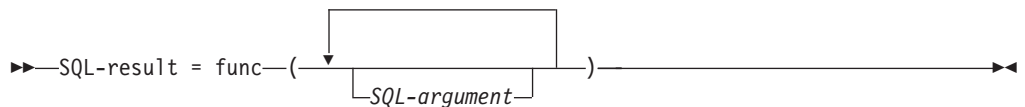
- | 0 This is a *normal call*. (All the normal input argument values are passed).
- | 1 This is a *final call*. No *SQL-argument* or *SQL-argument-ind* values are passed. A UDF should not return any answer using the *SQL-result*, *SQL-result-ind* arguments, *SQL-state*, or *diagnostic-message* arguments. These arguments are ignored by DB2 upon return from the UDF.

For table functions:

- | -2 This is the *first call* to the UDF for this statement. A first call is a *normal call* in that all SQL argument values are passed.
- | -1 This is the *open call* to the UDF for this statement. The scratchpad is initialized if NO FINAL CALL is specified, but not necessarily otherwise. All SQL argument values are passed.
- | 0 This is a *fetch call*. DB2 expects the table function to return either a row comprising the set of return values, or an end-of-table condition indicated by *SQLSTATE* value '02000'.
- | 1 This is a *close call*. This call balances the OPEN call, and can be used to perform any external CLOSE processing and resource release.
- | 2 This is a *final call*. No *SQL-argument* or *SQL-argument-ind* values are passed. A UDF should not return any answer using the *SQL-result*, *SQL-result-ind* arguments, *SQL-state*, or *diagnostic-message* arguments. These arguments are ignored by DB2 upon return from the UDF.

dbinfo This argument is set by DB2 before calling the UDF. It is only present if the CREATE FUNCTION statement for the UDF specifies the DBINFO keyword. The argument is a structure whose definition is contained in the sqludf include.

Parameter Style GENERAL (or SIMPLE CALL): With parameter style GENERAL, the parameters are passed into the external service program just as they are specified in the CREATE FUNCTION statement. This parameter style can only be used with scalar UDFs. The format is:



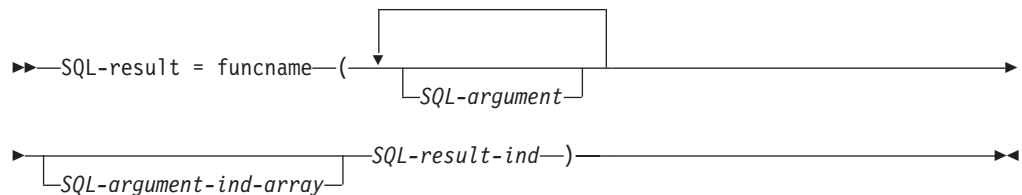
SQL-argument
 This argument is set by DB2 before calling the UDF. This value repeats *n* times, where *n* is the number of arguments specified in the function reference. The value of each of these arguments is taken from the expression specified in the function invocation. It is expressed in the data type of the defined parameter in the CREATE FUNCTION statement. Note: These parameters are treated as input only; any changes to the parameter values made by the UDF are ignored by DB2.

SQL-result
 This value is returned by the UDF. DB2 copies the value into database storage. In order to return the value correctly, the function code must be a value-returning function. The database copies only as much of the value as

defined for the return value as specified on the CREATE FUNCTION statement. If the CAST FROM clause is used in the CREATE FUNCTION statement, DB2 assumes the UDF returns the value as defined in the CAST FROM clause, otherwise DB2 assumes the UDF returns the value as defined in the RETURNS clause.

Because of the requirement that the function code be a value-returning function, any function code used for parameter style GENERAL must be created into a service program.

Parameter Style GENERAL WITH NULLS: The parameter style GENERAL WITH NULLS can only be used with scalar UDFs. With this parameter style, the parameters are passed into the service program as follows (in the order specified):



SQL-argument

This argument is set by DB2 before calling the UDF. This value repeats n times, where n is the number of arguments specified in the function reference. The value of each of these arguments is taken from the expression specified in the function invocation. It is expressed in the data type of the defined parameter in the CREATE FUNCTION statement. Note: These parameters are treated as input only; any changes to the parameter values made by the UDF are ignored by DB2.

SQL-argument-ind-array

This argument is set by DB2 before calling the UDF. It can be used by the UDF to determine if one or more *SQL-arguments* are null or not. It is an array of two-byte signed integers (indicators). The n th array argument corresponds to the n th *SQL-argument*. Each array entry is set to one of the following values:

- 0** The argument is present and not null.
- 1** The argument is null.

The UDF should check for null input. Note: This parameter is treated as input only; any changes to the parameter value made by the UDF is ignored by DB2.

SQL-result-ind

This argument is set by the UDF before returning to DB2. The database provides the storage for the return value. The argument is defined as a two-byte signed integer. If set to a negative value, the database interprets the result of the function as null. If set to zero or a positive value, the database uses the value returned in *SQL-result*. The database provides the storage for the return value indicator. Since the parameter is passed by address, the address is of the storage where the indicator value should be placed.

SQL-result

This value is returned by the UDF. DB2 copies the value into database storage. In order to return the value correctly, the function code must be a value-returning function. The database copies only as much of the value as defined for the return value as specified on the CREATE FUNCTION statement. If the CAST FROM clause is used in the CREATE FUNCTION statement, DB2 assumes the UDF returns the value as defined in the CAST FROM clause, otherwise DB2 assumes the UDF returns the value as defined in the RETURNS clause.

Because of the requirement that the function code be a value-returning function, any function code used for parameter style GENERAL WITH NULLS must be created into a service program.

Notes:

1. The external name specified on the CREATE FUNCTION statement can be specified either with quotes or without quotes. If the name is not quoted, it is uppercased before it is stored; if it is quoted, it is stored as specified. This becomes important when naming the actual program, as the database searches for the program that has a name that exactly matches the name stored with the function definition. For example, if a function was created as:

```
CREATE FUNCTION X(INT) RETURNS INT
LANGUAGE C
EXTERNAL NAME 'MYLIB/MYPGM(MYENTRY)'
```

and the source for the program was:

```
void myentry(
    int*in
    int*out,
    .
    .
    . .
```

the database would not find the entry because it is in lower case *myentry* and the database was instructed to look for uppercase *MYENTRY*.

2. For service programs with C++ modules, make sure in the C++ source code to precede the program function definition with *extern "C"*. Otherwise, the C++ compiler will perform 'name mangling' of the function's name and the database will not find it.

Parameter style DB2GENERAL: Parameter style DB2GENERAL is used by Java UDFs. For a description of this parameter style, see Java SQL Routines in the IBM Developer Kit for Java topic.

Parameter style Java: The Java parameter style is the style specified by the SQLJ Part 1: SQL Routines standard. For a description of this parameter style, see Java SQL Routines in the IBM Developer Kit for Java topic.

Table function considerations

An external table function is a UDF that delivers a table to the SQL in which it was referenced. A table function reference is only valid in a FROM clause of a SELECT. When using table functions, observe the following:

- Even though a table function delivers a table, the physical interface between DB2 and the UDF is one-row-at-a-time. There are five types of calls made to a table function: OPEN, FETCH, CLOSE, FIRST, and FINAL. The existence of

FIRST and FINAL calls depends on how you define the UDF. The same *call-type* mechanism that can be used for scalar functions is used to distinguish these calls.

- The standard interface used between DB2 and user-defined scalar functions is extended to accommodate table functions. The *SQL-result* argument repeats for table functions; each instance corresponding to a column to be returned as defined in the RETURNS TABLE clause of the CREATE FUNCTION statement. The *SQL-result-idx* argument likewise repeats, each instance related to the corresponding *SQL-result* instance.
- Not every result column defined in the RETURNS clause of the CREATE FUNCTION statement for the table function has to be returned. The DBINFO keyword of CREATE FUNCTION, and corresponding *dbinfo* argument enable the optimization that only those columns needed for a particular table function reference need be returned.
- The individual column values returned conform in format to the values returned by scalar functions.
- The CREATE FUNCTION statement for a table function has a CARDINALITY *n* specification. This specification enables the definer to inform the DB2 optimizer of the approximate size of the result so that the optimizer can make better decisions when the function is referenced. Regardless of what has been specified as the CARDINALITY of a table function, exercise caution against writing a function with infinite cardinality; that is, a function that always returns a row on a FETCH call. DB2 expects the *end-of-table* condition, as a catalyst within its query processing. So a table function that never returns the end-of-table condition (SQL-state value '02000') will cause an infinite processing loop.

Table function error processing

The error processing model for table function calls is as follows:

1. If FIRST call fails, no further calls are made.
2. If FIRST call succeeds, the nested OPEN, FETCH, and CLOSE calls are made, and the FINAL call is always made.
3. If OPEN call fails, no FETCH or CLOSE call is made.
4. If OPEN call succeeds, then FETCH and CLOSE calls are made.
5. If a FETCH call fails, no further FETCH calls are made, but the CLOSE call is made.

Note: This model describes the ordinary error processing for table UDFs. In the event of a system failure or communication problem, a call indicated by the error processing model may not be made.

Scalar function error processing

The error processing model for scalar UDFs which are defined with the FINAL CALL specification is as follows:

1. If FIRST call fails, no further calls are made.
2. If FIRST call succeeds, then further NORMAL calls are made as warranted by the processing of the statement, and a FINAL call is always made.
3. If NORMAL call fails, no further NORMAL calls are made, but the FINAL call is made (if you have specified FINAL CALL). This means that if an error is returned on a FIRST call, the UDF must clean up before returning, because no FINAL call will be made.

Note: This model describes the ordinary error processing for scalar UDFs. In the event of a system failure or communication problem, a call indicated by the error processing model may not be made.

Making a function fenced or unfenced

When creating a User Defined Function (UDF) consider whether to make the UDF an Unfenced UDF. By default, UDFs are created as Fenced UDFs. Fenced indicates that the database should run the UDF in a separate thread. For complex UDFs, this separation is meaningful as it will avoid potential problems such as generating unique SQL cursor names. Not having to be concerned about resource conflicts is one reason to stick with the default and create the UDF as a fenced UDF. A UDF created with the 'NOT FENCED' option indicates to the database that the user is requesting that the UDF can run within the same thread that initiated the UDF. Unfenced is a suggestion to the database, which could still decide to run the UDF in the same manner as a Fenced UDF.

```
CREATE FUNCTION QGPL.FENCED (parameter1 INTEGER)  
RETURNS INTEGER LANGUAGE SQL  
BEGIN  
RETURN parameter1 * 3;  
END;
```

```
CREATE FUNCTION QGPL.UNFENCED1 (parameter1 INTEGER)  
RETURNS INTEGER LANGUAGE SQL NOT FENCED  
-- Build the UDF to request faster execution via the NOT FENCED option  
BEGIN  
RETURN parameter1 * 3;  
END;
```

Examples of UDF code

These examples show how to implement UDF code by using SQL functions and external functions:

- "Example: Square of a number UDF"
- "Example: Counter" on page 244
- "Example: Weather table function" on page 244

Example: Square of a number UDF

Note: See "Code disclaimer information" on page x information for information pertaining to code examples.

Suppose that you wanted a function that returns the square of a number. The query statement is:

```
SELECT SQUARE(myint) FROM mytable
```

The following examples show how to define the UDF several different ways.

- **Using an SQL function**

```
CREATE FUNCTION SQUARE( inval INT) RETURNS INT  
LANGUAGE SQL  
BEGIN  
RETURN(inval*inval);  
END
```

- **Using an external function, parameter style SQL:**

The CREATE FUNCTION statement:


```

CREATE FUNCTION SQUARE(INT) RETURNS INT CAST FROM FLOAT
LANGUAGE C
EXTERNAL NAME 'MYLIB/MATH(SQUARE)'
DETERMINISTIC
NO SQL
NO EXTERNAL ACTION
PARAMETER STYLE SQL
ALLOW PARALLEL

```

The code:

```

void SQUARE(int *inval,
double *outval,
short *inind,
short *outind,
char *sqlstate,
char *funcname,
char *specname,
char *msgtext)
{
if (*inind<0)
*outind=-1;
else
{
*outval=*inval;
*outval=(*outval)*(*outval);
*outind=0;
}
return;
}

```

To create the external service program so it can be debugged:

```

CRTCMOD MODULE(mylib/square) DBGVIEW(*SOURCE)
CRTSRVPGM SRVPGM(mylib/math) MODULE(mylib/square)
EXPORT(*ALL) ACTGRP(*CALLER)

```

- **Using an external function, parameter style GENERAL:**

The CREATE FUNCTION statement:

```

CREATE FUNCTION SQUARE(INT) RETURNS INT CAST FROM FLOAT
LANGUAGE C
EXTERNAL NAME 'MYLIB/MATH(SQUARE)'
DETERMINISTIC
NO SQL
NO EXTERNAL ACTION
PARAMETER STYLE GENERAL
ALLOW PARALLEL

```

The code:

```

double SQUARE(int *inval)
{
double outval;
outval=*inval;
outval=outval*outval;
return(outval);
}

```

To create the external service program so it can be debugged:

```

CRTCMOD MODULE(mylib/square) DBGVIEW(*SOURCE)

CRTSRVPGM SRVPGM(mylib/math) MODULE(mylib/square)
EXPORT(*ALL) ACTGRP(*CALLER)

```

Example: Counter

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Suppose you want to simply number the rows in your SELECT statement. So you write a UDF which increments and returns a counter. This example uses an external function with DB2 SQL parameter style and a scratchpad.

```
CREATE FUNCTION COUNTER()
  RETURNS INT
  SCRATCHPAD
  NOT DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'MYLIB/MATH(ctr)'
  DISALLOW PARALLEL

/* structure scr defines the passed scratchpad for the function "ctr" */
struct scr {
  long len;
  long countr;
  char not_used[96];
};

void ctr (
  long *out,                /* output answer (counter) */
  short *outnull,          /* output NULL indicator */
  char *sqlstate,          /* SQL STATE */
  char *funcname,          /* function name */
  char *specname,          /* specific function name */
  char *mesgtext,          /* message text insert */
  struct scr *scratchptr) { /* scratch pad */

  *out = ++scratchptr->countr; /* increment counter & copy out */
  *outnull = 0;
  return;
}
/* end of UDF : ctr */
```

For this UDF, observe that:

- It has no input SQL arguments defined, but returns a value.
- It appends the scratchpad input argument after the four standard trailing arguments, namely *SQL-state*, *function-name*, *specific-name*, and *message-text*.
- It includes a structure definition to map the scratchpad which is passed.
- No input parameters are defined. This agrees with the code.
- SCRATCHPAD is coded, causing DB2 to allocate, properly initialize and pass the scratchpad argument.
- You have specified it to be NOT DETERMINISTIC, because it depends on more than the SQL input arguments, (none in this case).
- You have correctly specified DISALLOW PARALLEL, because correct functioning of the UDF depends on a single scratchpad.

Example: Weather table function

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

The following is an example table function that returns weather information for various cities in the United States. The weather data for these cities is read in from

an external file, as indicated in the comments contained in the example program. The data includes the name of a city followed by its weather information. This pattern is repeated for the other cities.

```

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h> /* for use in compiling User Defined Function */

#define SQL_NOTNULL 0 /* Nulls Allowed - Value is not Null */
#define SQL_ISNULL -1 /* Nulls Allowed - Value is Null */
#define SQL_TYP_VARCHAR 448
#define SQL_TYP_INTEGER 496
#define SQL_TYP_FLOAT 480

/* Short and long city name structure */
typedef struct {
    char * city_short ;
    char * city_long ;
} city_area ;

/* Scratchpad data */ 1
/* Preserve information from one function call to the next call */
typedef struct {
    /* FILE * file_ptr; if you use weather data text file */
    int file_pos ; /* if you use a weather data buffer */
} scratch_area ;

/* Field descriptor structure */
typedef struct {
    char fld_field[31] ; /* Field data */
    int fld_ind ; /* Field null indicator data */
    int fld_type ; /* Field type */
    int fld_length ; /* Field length in the weather data */
    int fld_offset ; /* Field offset in the weather data */
} fld_desc ;

/* Short and long city name data */
city_area cities[] = {
    { "alb", "Albany, NY" },
    { "atl", "Atlanta, GA" },
    .
    .
    .
    { "wbc", "Washington DC, DC" },
    /* You may want to add more cities here */

    /* Do not forget a null termination */
    { ( char * ) 0, ( char * ) 0 }
} ;

/* Field descriptor data */
fld_desc fields[] = {
    { "", SQL_ISNULL, SQL_TYP_VARCHAR, 30, 0 }, /* city */
    { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 2 }, /* temp_in_f */
    { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 7 }, /* humidity */
    { "", SQL_ISNULL, SQL_TYP_VARCHAR, 5, 13 }, /* wind */
    { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 19 }, /* wind_velocity */
    { "", SQL_ISNULL, SQL_TYP_FLOAT, 5, 24 }, /* barometer */
    { "", SQL_ISNULL, SQL_TYP_VARCHAR, 25, 30 }, /* forecast */
    /* You may want to add more fields here */

    /* Do not forget a null termination */
    { ( char ) 0, 0, 0, 0, 0 }
} ;

```

```

/* Following is the weather data buffer for this example. You */
/* may want to keep the weather data in a separate text file. */
/* Uncomment the following fopen() statement. Note that you */
/* have to specify the full path name for this file.          */
char * weather_data[] = {
    "alb.forecast",
    " 34 28% wnw 3 30.53 clear",
    "atl.forecast",
    " 46 89% east 11 30.03 fog",
    .
    .
    .
    "wbc.forecast",
    " 38 96% ene 16 30.31 light rain",
    /* You may want to add more weather data here */

    /* Do not forget a null termination */
    ( char * ) 0
} ;

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Find a full city name using a short name */
int get_name( char * short_name, char * long_name ) {

    int name_pos = 0 ;

    while ( cities[name_pos].city_short != ( char * ) 0 ) {
        if ( strcmp( short_name, cities[name_pos].city_short ) == 0 ) {
            strcpy( long_name, cities[name_pos].city_long ) ;
            /* A full city name found */
            return( 0 ) ;
        }
        name_pos++ ;
    }
    /* Could not find such city in the city data */
    strcpy( long_name, "Unknown City" ) ;
    return( -1 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Clean all field data and field null indicator data */
int clean_fields( int field_pos ) {

    while ( fields[field_pos].fld_length !=0 ) {
        memset( fields[field_pos].fld_field, '\0', 31 ) ;
        fields[field_pos].fld_ind = SQL_ISNULL ;
        field_pos++ ;
    }
    return( 0 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Fills all field data and field null indicator data ... */
/* ... from text weather data */
int get_value( char * value, int field_pos ) {

```

```

fld_desc * field ;
char field_buf[31] ;
double * double_ptr ;
int * int_ptr, buf_pos ;

while ( fields[field_pos].fld_length != 0 ) {
    field = &fields[field_pos] ;
    memset( field_buf, '\0', 31 ) ;
    memcpy( field_buf,
           ( value + field->fld_offset ),
           field->fld_length ) ;
    buf_pos = field->fld_length ;
    while ( ( buf_pos > 0 ) &&
           ( field_buf[buf_pos] == ' ' ) )
        field_buf[buf_pos--] = '\0' ;
    buf_pos = 0 ;
    while ( ( buf_pos < field->fld_length ) &&
           ( field_buf[buf_pos] == ' ' ) )
        buf_pos++ ;
    if ( strlen( ( char * ) ( field_buf + buf_pos ) ) > 0 ||
        strcmp( ( char * ) ( field_buf + buf_pos ), "n/a" ) != 0 ) {
        field->fld_ind = SQL_NOTNULL ;

        /* Text to SQL type conversion */
        switch( field->fld_type ) {
            case SQL_TYP_VARCHAR:
                strcpy( field->fld_field,
                       ( char * ) ( field_buf + buf_pos ) ) ;
                break ;
            case SQL_TYP_INTEGER:
                int_ptr = ( int * ) field->fld_field ;
                *int_ptr = atoi( ( char * ) ( field_buf + buf_pos ) ) ;
                break ;
            case SQL_TYP_FLOAT:
                double_ptr = ( double * ) field->fld_field ;
                *double_ptr = atof( ( char * ) ( field_buf + buf_pos ) ) ;
                break ;
        }
        /* You may want to add more text to SQL type conversion here */
    }

    field_pos++ ;
}
return( 0 ) ;
}

#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN weather( /* Return row fields */
                        SQLUDF_VARCHAR * city,
                        SQLUDF_INTEGER * temp_in_f,
                        SQLUDF_INTEGER * humidity,
                        SQLUDF_VARCHAR * wind,
                        SQLUDF_INTEGER * wind_velocity,
                        SQLUDF_DOUBLE * barometer,
                        SQLUDF_VARCHAR * forecast,
                        /* You may want to add more fields here */

                        /* Return row field null indicators */
                        SQLUDF_NULLIND * city_ind,
                        SQLUDF_NULLIND * temp_in_f_ind,
                        SQLUDF_NULLIND * humidity_ind,
                        SQLUDF_NULLIND * wind_ind,
                        SQLUDF_NULLIND * wind_velocity_ind,

```

```

        SQLUDF_NULLIND * barometer_ind,
        SQLUDF_NULLIND * forecast_ind,
        /* You may want to add more field indicators here */

        /* UDF always-present (trailing) input arguments */
        SQLUDF_TRAIL_ARGS_ALL
    ) {

scratch_area * save_area ;
char line_buf[81] ;
int line_buf_pos ;

/* SQLUDF_SCRAT is part of SQLUDF_TRAIL_ARGS_ALL */
/* Preserve information from one function call to the next call */
save_area = ( scratch_area * ) ( SQLUDF_SCRAT->data ) ;

/* SQLUDF_CALLT is part of SQLUDF_TRAIL_ARGS_ALL */
switch( SQLUDF_CALLT ) {

    /* First call UDF: Open table and fetch first row */
    case SQL_TF_OPEN:
        /* If you use a weather data text file specify full path */
        /* save_area->file_ptr = fopen("tblsrv.dat","r"); */
        save_area->file_pos = 0 ;
        break ;

    /* Normal call UDF: Fetch next row */ 2
    case SQL_TF_FETCH:
        /* If you use a weather data text file */
        /* memset(line_buf, '\0', 81); */
        /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
        if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {

            /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
            strcpy( SQLUDF_STATE, "02000" ) ;

            break ;
        }
        memset( line_buf, '\0', 81 ) ;
        strcpy( line_buf, weather_data[save_area->file_pos] ) ;
        line_buf[3] = '\0' ;

        /* Clean all field data and field null indicator data */
        clean_fields( 0 ) ;

        /* Fills city field null indicator data */
        fields[0].fld_ind = SQL_NOTNULL ;

        /* Find a full city name using a short name */
        /* Fills city field data */
        if ( get_name( line_buf, fields[0].fld_field ) == 0 ) {
            save_area->file_pos++ ;
            /* If you use a weather data text file */
            /* memset(line_buf, '\0', 81); */
            /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
            if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {
                /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
                strcpy( SQLUDF_STATE, "02000" ) ;
                break ;
            }
            memset( line_buf, '\0', 81 ) ;
            strcpy( line_buf, weather_data[save_area->file_pos] ) ;
            line_buf_pos = strlen( line_buf ) ;
            while ( line_buf_pos > 0 ) {
                if ( line_buf[line_buf_pos] >= ' ' )
                    line_buf_pos = 0 ;
                else {

```

```

        line_buf[line_buf_pos] = '\0' ;
        line_buf_pos-- ;
    }
}

/* Fills field data and field null indicator data ... */
/* ... for selected city from text weather data */
get_value( line_buf, 1 ) ; /* Skips city field */

/* Builds return row fields */
strcpy( city, fields[0].fld_field ) ;
memcpy( (void *) temp_in_f,
        fields[1].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
memcpy( (void *) humidity,
        fields[2].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
strcpy( wind, fields[3].fld_field ) ;
memcpy( (void *) wind_velocity,
        fields[4].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
memcpy( (void *) barometer,
        fields[5].fld_field,
        sizeof( SQLUDF_DOUBLE ) ) ;
strcpy( forecast, fields[6].fld_field ) ;

/* Builds return row field null indicators */
memcpy( (void *) city_ind,
        &(fields[0].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) temp_in_f_ind,
        &(fields[1].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) humidity_ind,
        &(fields[2].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) wind_ind,
        &(fields[3].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) wind_velocity_ind,
        &(fields[4].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) barometer_ind,
        &(fields[5].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) forecast_ind,
        &(fields[6].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;

/* Next city weather data */
save_area->file_pos++ ;

break ;

/* Special last call UDF for cleanup (no real args!): Close table */ 3
case SQL_IF_CLOSE:
/* If you use a weather data text file */
/* fclose(save_area->file_ptr); */
/* save_area->file_ptr = NULL; */
save_area->file_pos = 0 ;
break ;
}
}

```

Referring to the embedded numbers in this UDF code, observe that:

1. The scratchpad is defined. The row variable is initialized on the OPEN call, and the iptr array and nbr_rows variable are filled in by the *mystery* function at open time.
2. FETCH traverses the iptr array, using row as an index, and moves the values of interest from the current element of iptr to the location pointed to by out_c1, out_c2, and out_c3 result value pointers.
3. Finally, CLOSE frees the storage acquired by OPEN and anchored in the scratchpad.

Following is the CREATE FUNCTION statement for this UDF:

```
CREATE FUNCTION tfweather_u()  
  RETURNS TABLE (CITY VARCHAR(25),  
                 TEMP_IN_F INTEGER,  
                 HUMIDITY INTEGER,  
                 WIND VARCHAR(5),  
                 WIND_VELOCITY INTEGER,  
                 BAROMETER FLOAT,  
                 FORECAST VARCHAR(25))  
  
  SPECIFIC tfweather_u  
  DISALLOW PARALLEL  
  NOT FENCED  
  DETERMINISTIC  
  NO SQL  
  NO EXTERNAL ACTION  
  SCRATCHPAD  
  NO FINAL CALL  
  LANGUAGE C  
  PARAMETER STYLE DB2SQL  
  EXTERNAL NAME 'LIB1/WEATHER(weather)';
```

Referring to this statement, observe that:

- It does not take any input, and returns 7 output columns.
- SCRATCHPAD is specified, so DB2 allocates, properly initializes and passes the scratchpad argument.
- NO FINAL CALL is specified.
- The function is specified as NOT DETERMINISTIC, because it depends on more than the SQL input arguments. That is, it depends on the mystery function and we assume that the content can vary from execution to execution.
- DISALLOW PARALLEL is required for table functions.
- CARDINALITY 100 is an estimate of the expected number of rows returned, provided to the DB2 optimizer.
- DBINFO is not used, and the optimization to only return the columns needed by the particular statement referencing the function is not implemented.
- NOT NULL CALL is specified, so the UDF will not be called if any of its input SQL arguments are NULL, and does not have to check for this condition.

To select all of the rows generated by this table function, use the following query:

```
SELECT *  
  FROM TABLE (tfweather_u())x
```

Chapter 14. Dynamic SQL Applications

Dynamic SQL allows an application to define and run SQL statements at program run time. An application that provides for dynamic SQL accepts as input (or builds) an SQL statement in the form of a character string. The application does not need to know what type of SQL statement it will run. The application:

- Builds or accepts as input an SQL statement
- Prepares the SQL statement for running
- Runs the statement
- Handles SQL return codes

Interactive SQL (described in Chapter 17, “Using Interactive SQL”) is an example of a dynamic SQL program. SQL statements are processed and run dynamically by interactive SQL.

Notes:

1. The run-time overhead is greater for statements processed using dynamic SQL than for static SQL statements. The additional process is similar to that required for precompiling, binding, and then running a program, instead of only running it. Therefore, only applications requiring the flexibility of dynamic SQL should use it. Other applications should access data from the database using normal (static) SQL statements.
2. Programs that contain an EXECUTE or EXECUTE IMMEDIATE statement and that use a FOR READ ONLY clause to make a cursor read-only experience better performance because blocking is used to retrieve rows for the cursor. The ALWBLK(*ALLREAD) CRTSQLxxx option will imply a FOR READ ONLY declaration for all cursors that do not explicitly code FOR UPDATE OF or have positioned deletes or updates that refer to the cursor. Cursors with an implied FOR READ ONLY will benefit from the second item in this list.

Some dynamic SQL statements require use of address variables. RPG for iSeries programs require the aid of PL/I, COBOL, C, or ILE RPG for iSeries programs to manage the address variables.

The following table shows all the statements supported by DB2 UDB for iSeries and indicates if they can be used in a dynamic application.

Note: In the following table, the numbers in the *Dynamic SQL* column correspond to the notes on the next page.

Table 28. List of SQL Statements Allowed in Dynamic Applications

SQL Statement	Static SQL	Dynamic SQL
ALTER TABLE	Y	Y
BEGIN DECLARE SECTION	Y	N
CALL	Y	Y
CLOSE	Y	N
COMMENT ON	Y	Y
COMMIT	Y	Y
CONNECT	Y	N

Table 28. List of SQL Statements Allowed in Dynamic Applications (continued)

SQL Statement	Static SQL	Dynamic SQL
CREATE ALIAS	Y	Y
CREATE DISTINCT TYPE	Y	Y
CREATE FUNCTION	Y	Y
CREATE INDEX	Y	Y
CREATE PROCEDURE	Y	Y
CREATE SCHEMA	Y	Y
CREATE TABLE	Y	Y
CREATE TRIGGER	Y	Y
CREATE VIEW	Y	Y
DECLARE CURSOR	Y	See Note 1.
DECLARE GLOBAL TEMPORARY TABLE	Y	Y
DECLARE PROCEDURE	Y	N
DECLARE STATEMENT	Y	N
DECLARE VARIABLE	Y	N
DELETE	Y	Y
DESCRIBE	Y	See Note 2.
DESCRIBE TABLE	Y	N
DISCONNECT	Y	N
DROP	Y	Y
END DECLARE SECTION	Y	N
EXECUTE	Y	See Note 3.
EXECUTE IMMEDIATE	Y	See Note 4.
FETCH	Y	N
FREE LOCATOR	Y	Y
GRANT	Y	Y
HOLD LOCATOR	Y	Y
INCLUDE	Y	N
INSERT	Y	Y
LABEL ON	Y	Y
LOCK TABLE	Y	Y
OPEN	Y	N
PREPARE	Y	See Note 5.
RELEASE	Y	N
RELEASE SAVEPOINT	Y	Y
RENAME	Y	Y
REVOKE	Y	Y
ROLLBACK	Y	Y
SAVEPOINT	Y	Y
SELECT INTO	Y	See Note 6.
SELECT statement	Y	See Note 7.

Table 28. List of SQL Statements Allowed in Dynamic Applications (continued)

SQL Statement	Static SQL	Dynamic SQL
SET CONNECTION	Y	N
SET OPTION	Y	See Note 8.
SET PATH	Y	Y
SET RESULT SETS	Y	N
SET SCHEMA	Y	Y
SET TRANSACTION	Y	Y
SET variable	Y	N
UPDATE	Y	Y
VALUES INTO	Y	N
WHENEVER	Y	N

Notes:

1. Cannot be prepared, but used to define the cursor for the associated dynamic SELECT statement prior to running.
2. Cannot be prepared, but used to return a description of a prepared statement.
3. Cannot be prepared, but used to run prepared SQL statements. The SQL statement must be previously prepared by the PREPARE statement prior to using the EXECUTE statement. See example for PREPARE under “Using the PREPARE and EXECUTE statements” on page 254.
4. Cannot be prepared, but used with dynamic statement strings that do not have any ? parameter markers. The EXECUTE IMMEDIATE statement causes the statement strings to be prepared and run dynamically at program run time. See example for EXECUTE IMMEDIATE under “Processing non-SELECT statements” on page 254.
5. Cannot be prepared, but used to parse, optimize, and set up dynamic SELECT statements prior to running. See example for PREPARE under “Processing non-SELECT statements” on page 254.
6. A SELECT INTO statement cannot be prepared or used in EXECUTE IMMEDIATE.
7. Cannot be used with EXECUTE or EXECUTE IMMEDIATE but can be prepared and used with OPEN.
8. Can only be used when running a REXX procedure or in a precompiled program.

Note: See “Code disclaimer information” on page x information for information pertaining to code examples.

Designing and running a dynamic SQL application

To issue a dynamic SQL statement, you must use the statement with either an EXECUTE statement or an EXECUTE IMMEDIATE statement, because dynamic SQL statements are not prepared at precompile time and therefore must be prepared at run time. The EXECUTE IMMEDIATE statement causes the SQL statement to be prepared and run dynamically at program run time.

There are two basic types of dynamic SQL statements: SELECT statements and non-SELECT statements. Non-SELECT statements include such statements as DELETE, INSERT, and UPDATE.

Client server applications that use interfaces such as ODBC typically use dynamic SQL to access the database. For more information about developing client server applications that use iSeries Access, see Programming for iSeries Access Express.

Processing non-SELECT statements

To build a dynamic SQL non-SELECT statement:

1. Verify that the SQL statement you want to build is one that can be run dynamically (see Table 28 on page 251).
2. Build the SQL statement. (Use Interactive SQL for an easy way to build, verify, and run your SQL statement. See Chapter 17, "Using Interactive SQL" for more information.)

To run a dynamic SQL non-SELECT statement:

1. Run the SQL statement using EXECUTE IMMEDIATE, or PREPARE the SQL statement, then EXECUTE the prepared statement.
2. Handle any SQL return codes that might result.

The following is an example of an application running a dynamic SQL non-SELECT statement (stmtstrg):

```
EXEC SQL  
EXECUTE IMMEDIATE :stmtstrg;
```

CCSID of dynamic SQL statements

The SQL statement is normally a host variable. The CCSID of the host variable is used as the CCSID of the statement text. In PL/I, it also can be a string expression. In this case, the job CCSID is used as the CCSID of the statement text.

Dynamic SQL statements are processed using the CCSID of the statement text. This affects variant characters the most. For example, the not sign (¬) is located at 'BA'X in CCSID 500. This means that if the CCSID of your statement text is 500, SQL expects the not sign (¬) to be located at 'BA'X.

If the statement text CCSID is 65535, SQL processes variant characters as if they had a CCSID of 37. This means that SQL looks for the not sign (¬) at '5F'X.

Using the PREPARE and EXECUTE statements

If non-SELECT statements contain no parameter markers, they can be run dynamically using the EXECUTE IMMEDIATE statement. However, if the non-SELECT statements have parameter markers, they must be run using PREPARE and EXECUTE.

The PREPARE statement prepares the non-SELECT statement (for example, the DELETE statement) and gives it a name of your choosing. If DLYPRP (*YES) is specified on the CRTSQLxxx command, the preparation is delayed until the first time the statement is used in an EXECUTE or DESCRIBE statement, unless the USING clause is specified on the PREPARE statement. In this instance, let us call it S1. After the statement has been prepared, it can be run many times within the same program, using different values for the parameter markers. The following example is of a prepared statement being run multiple times:

```
DSTRING = 'DELETE FROM CORPDATA.EMPLOYEE WHERE EMPNO = ?';
```

```
/*The ? is a parameter marker which denotes  
that this value is a host variable that is
```

```

    to be substituted each time the statement is run.*/

EXEC SQL PREPARE S1 FROM :DSTRING;

/*DSTRING is the delete statement that the PREPARE statement is
naming S1.*/

DO UNTIL (EMP =0);
/*The application program reads a value for EMP from the
display station.*/
EXEC SQL
    EXECUTE S1 USING :EMP;

END;

```

In routines similar to the example above, you must know the number of parameter markers and their data types, because the host variables that provide the input data are declared when the program is being written.

Note: All prepared statements that are associated with an application server are destroyed whenever the connection to the application server ends. Connections are ended by a CONNECT (Type 1) statement, a DISCONNECT statement, or a RELEASE followed by a successful COMMIT.

Processing SELECT statements and using an SQLDA

There are two basic types of SELECT statements: **fixed-list** and **varying-list**.

To process a fixed-list SELECT statement, an SQLDA is not necessary.

To process a varying-list SELECT statement, you must first declare an SQLDA structure. The SQLDA is a control block used to pass host variable input values from an application program to SQL and to receive output values from SQL. In addition, information about SELECT list expressions can be returned in a PREPARE or DESCRIBE statement.

Fixed-list SELECT statements

In dynamic SQL, fixed-list SELECT statements are those statements designed to retrieve data of a predictable number and type. When using these statements, you can anticipate and define host variables to accommodate the retrieved data, so that an SQLDA is not necessary. Each successive FETCH returns the same number of values as the last, and these values have the same data formats as those returned for the last FETCH. You can specify host variables the same as you would for any SQL application.

You can use fixed-list dynamic SELECT statements with any SQL-supported application program.

To run fixed-list SELECT statements dynamically, your application must:

1. Place the input SQL statement into a host variable.
2. Issue a PREPARE statement to validate the dynamic SQL statement and put it into a form that can be run. If DLYPRP (*YES) is specified on the CRTSQLxxx command, the preparation is delayed until the first time the statement is used in an EXECUTE or DESCRIBE statement, unless the USING clause is specified on the PREPARE statement.
3. Declare a cursor for the statement name.

4. Open the cursor.
5. FETCH a row into a fixed list of variables (rather than into a descriptor area, as you would if you were using a varying-list SELECT statement, described in the following section, Varying-list Select-statements).
6. When end of data occurs, close the cursor.
7. Handle any SQL return codes that result.

For example:

```
MOVE 'SELECT EMPNO, LASTNAME FROM CORPDATA.EMPLOYEE WHERE EMPNO>?'
TO DSTRING.
EXEC SQL
  PREPARE S2 FROM :DSTRING END-EXEC.

EXEC SQL
  DECLARE C2 CURSOR FOR S2 END-EXEC.

EXEC SQL
  OPEN C2 USING :EMP END-EXEC.

PERFORM FETCH-ROW UNTIL SQLCODE NOT=0.

EXEC SQL
  CLOSE C2 END-EXEC.
STOP-RUN.
FETCH-ROW.
EXEC SQL
  FETCH C2 INTO :EMP, :EMPNAME END-EXEC.
```

Note: Remember that because the SELECT statement, in this case, always returns the same number and type of data items as previously run fixed-list SELECT statements, you do not have to use the SQL descriptor area (SQLDA).

Varying-list Select-statements

In dynamic SQL, varying-list SELECT statements are ones for which the number and format of result columns to be returned are not predictable; that is, you do not know how many variables you need, or what the data types are. Therefore, you cannot define host variables in advance to accommodate the result columns returned.

Note: In REXX, steps 5.b, 6, and 7 are not applicable.

If your application accepts varying-list SELECT statements, your program has to:

1. Place the input SQL statement into a host variable.
2. Issue a PREPARE statement to validate the dynamic SQL statement and put it into a form that can be run. If DLYPRP (*YES) is specified on the CRTSQLxxx command, the preparation is delayed until the first time the statement is used in an EXECUTE or DESCRIBE statement, unless the USING clause is specified on the PREPARE statement.
3. Declare a cursor for the statement name.
4. Open the cursor (declared in step 3) that includes the name of the dynamic SELECT statement.
5. Issue a DESCRIBE statement to request information from SQL about the type and size of each column of the result table.

Notes:

- a. You can also code the PREPARE statement with an INTO clause to perform the functions of PREPARE and DESCRIBE with a single statement.
 - b. If the SQLDA is not large enough to contain column descriptions for each retrieved column, the program must determine how much space is needed, get storage for that amount of space, build a new SQLDA, and reissue the DESCRIBE statement.
6. Allocate the amount of storage needed to contain a row of retrieved data.
 7. Put storage addresses into the SQLDA (SQL descriptor area) to tell SQL where to put each item of retrieved data.
 8. FETCH a row.
 9. When end of data occurs, close the cursor.
 10. Handle any SQL return codes that might result.

See “Example: Select-statement for allocating storage for SQLDA” on page 262 for details on how to perform these steps.

SQL Descriptor Area (SQLDA)

Dynamic SQL uses a structure of variables called the SQL descriptor area (SQLDA) to pass information about an SQL statement between SQL and your application. The SQLDA is required for running the DESCRIBE and DESCRIBE TABLE statements, and can also be used on the PREPARE, OPEN, FETCH, CALL, and EXECUTE statements.

The meaning of the information in an SQLDA depends on its use. In PREPARE and DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In DESCRIBE TABLE, the SQLDA provides information to an application program about the columns in a table or view. In OPEN, EXECUTE, CALL, and FETCH, an SQLDA provides information about host variables. For example, you can read values into the SQLDA using a DESCRIBE statement, change the values with the addresses of host variables, and then reuse the values in a FETCH statement.

If your application allows you have several cursors open at the same time, you can code several SQLDAs, one for each dynamic SELECT statement. For more information, see SQLDA and SQLCA in the SQL Reference book.

SQLDAs can be used in C, C++, COBOL, PL/I, REXX, and RPG. Because RPG for iSeries does not provide a way to set pointers, the SQLDA must be set outside the RPG for iSeries program by a PL/I, C, C++, COBOL, or ILE RPG for iSeries program. That program must then call the RPG for iSeries program.

SQLDA format

The SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of six variables collectively named SQLVAR.

Note: The SQLDA in REXX is different. For more information, see the topic Coding SQL Statements in REXX Applications in the *SQL Programming with Host Languages* information.

When an SQLDA is used in OPEN, FETCH, CALL, and EXECUTE, each occurrence of SQLVAR describes a host variable.

The fields of the SQLDA are as follows:

SQLDAID

SQLDAID is as used an 'eyecatcher' for storage dumps. It is a string of 8 characters that have the value 'SQLDA' after the SQLDA is used in a PREPARE or DESCRIBE statement. This variable is not used for FETCH, OPEN, CALL, or EXECUTE.

Byte 7 can be used to determine if more than one SQLVAR entry is needed for each column. Multiple SQLVAR entries may be needed if there are any LOB or distinct type columns. This flag is set to a blank if there are not any LOBs or distinct types.

SQLDAID is not applicable in REXX.

SQLDABC

SQLDABC indicates the length of the SQLDA. It is a 4-byte integer that has the value $SQLN * LENGTH(SQLVAR) + 16$ after the SQLDA is used in a PREPARE or DESCRIBE statement. SQLDABC must have a value equal to or greater than $SQLN * LENGTH(SQLVAR) + 16$ prior to use by FETCH, OPEN, CALL, or EXECUTE.

SQLABC is not applicable in REXX.

SQLN SQLN is a 2-byte integer that specifies the total number of occurrences of SQLVAR. It must be set prior to use by any SQL statement to a value greater than or equal to 0.

SQLN is not applicable in REXX.

SQLD SQLD is a 2-byte integer that specifies the number of occurrences of SQLVAR, in other words, the number of host variables or columns described by the SQLDA. This field is set by SQL on a DESCRIBE or PREPARE statement. In other statements, this field must be set prior to use to a value greater than or equal to 0 and less than or equal to SQLN.

SQLVAR

This group of values are repeated once for each host variable or column. These variables are set by SQL on a DESCRIBE or PREPARE statement. In other statements, they must be set prior to use. These variables are defined as follows:

SQLTYPE

SQLTYPE is a 2-byte integer that specifies the data type of the host variable or column as shown in the Table 29 on page 259. Odd values for SQLTYPE show that the host variable has an associated indicator variable addressed by SQLIND.

SQLLEN

SQLLEN is a 2-byte integer variable that specifies the length attribute of the host variable or column.

SQLRES

SQLRES is a 12-byte reserved area for boundary alignment purposes. Note that, in OS/400, pointers *must* be on a quad-word boundary.

SQLRES is not applicable in REXX.

SQLDATA

SQLDATA is a 16-byte pointer variable that specifies the address of the host variables when the SQLDA is used on OPEN, FETCH, CALL, and EXECUTE.

When the SQLDA is used on PREPARE and DESCRIBE, this area is overlaid with the following information:

The CCSID of a character or graphic field is stored in the third and fourth bytes of SQLDATA. For BIT data, the CCSID is 65535. In REXX, the CCSID is returned in the variable SQLCCSID.

SQLIND

SQLIND is a 16-byte pointer that specifies the address of a small integer host variable that is used as an indication of null or not null when the SQLDA is used on OPEN, FETCH, CALL, and EXECUTE. A negative value indicates null and a non-negative indicates not null. This pointer is only used if SQLTYPE contains an odd value.

When the SQLDA is used on PREPARE and DESCRIBE, this area is reserved for future use.

SQLNAME

SQLNAME is a variable-length character variable with a maximum length of 30. After a PREPARE or DESCRIBE, this variable contains the name of selected column, label, or system column name. In OPEN, FETCH, EXECUTE, or CALL, this variable can be used to pass the CCSID of character strings. CCSIDs can be passed for character and graphic host variables.

The SQLNAME field in an SQLVAR array entry of an input SQLDA can be set to specify the CCSID:

Data Type	Sub-type	Length of SQLNAME	SQLNAME Bytes 1 & 2	SQLNAME Bytes 3 & 4
Character	SBCS	8	X'0000'	CCSID
Character	MIXED	8	X'0000'	CCSID
Character	BIT	8	X'0000'	X'FFFF'
GRAPHIC	not applicable	8	X'0000'	CCSID
Any other data type	not applicable	not applicable	not applicable	not applicable

Note: It is important to remember that the SQLNAME field is only for overriding the CCSID. Applications that use the defaults do not need to pass CCSID information. If a CCSID is not passed, the default CCSID for the job is used.

The default for graphic host variables is the associated double-byte CCSID for the job CCSID. If an associated double-byte CCSID does not exist, 65535 is used.

Table 29. SQLTYPE and SQLLEN Values for PREPARE, DESCRIBE, FETCH, OPEN, CALL, or EXECUTE

SQLTYPE	For PREPARE and DESCRIBE		For FETCH, OPEN, CALL, and EXECUTE	
	COLUMN DATA TYPE	SQLLEN	HOST VARIABLE DATA TYPE	SQLLEN
384/385	Date	10	Fixed-length character string representation of a date	Length attribute of the host variable
388/389	Time	8	Fixed-length character string representation of a time	Length attribute of the host variable

Table 29. *SQLTYPE* and *SQLLEN* Values for *PREPARE*, *DESCRIBE*, *FETCH*, *OPEN*, *CALL*, or *EXECUTE* (continued)

SQLTYPE	For PREPARE and DESCRIBE		For FETCH, OPEN, CALL, and EXECUTE	
	COLUMN DATA TYPE	SQLLEN	HOST VARIABLE DATA TYPE	SQLLEN
392/393	Timestamp	26	Fixed-length character string representation of a timestamp	Length attribute of the host variable
396/397	DataLink Note #1	Length attribute of the column	N/A	N/A
400/401	N/A	N/A	NUL-terminated graphic string	Length attribute of the host variable
392/393	Timestamp	26	Fixed-length character string representation of a timestamp	Length attribute of the host variable
404/405	BLOB	0 (See Note #2)	BLOB	Not used. (See Note #2)
408/409	CLOB	0 (See Note #2)	CLOB	Not used. (See Note #2)
412/413	DBCLOB	0 (See Note #2)	DBCLOB	Not used. (See Note #2)
452/453	Fixed-length character string	Length attribute of the column	Fixed-length character string	Length attribute of the host variable
456/457	Long varying-length character string	Length attribute of the column	Long varying-length character string	Length attribute of the host variable
460/461	N/A	N/A	NUL-terminated character string	Length attribute of the host variable
464/465	Varying-length graphic string	Length attribute of the column	Varying-length graphic string	Length attribute of the host variable
468/469	Fixed-length graphic string	Length attribute of the column	Fixed-length graphic string	Length attribute of the host variable
472/473	Long varying-length graphic string	Length attribute of the column	Long graphic string	Length attribute of the host variable
476/477	N/A	N/A	PASCAL L-string	Length attribute of the host variable
480/481	Floating point	4 for single precision, 8 for double precision	Floating point	4 for single precision, 8 for double precision
484/485	Packed decimal	Precision in byte 1; scale in byte 2	Packed decimal	Precision in byte 1; scale in byte 2
488/489	Zoned decimal	Precision in byte 1; scale in byte 2	Zoned decimal	Precision in byte 1; scale in byte 2
492/493	Big integer	8	Big integer	8

Table 29. *SQLTYPE and SQLLEN Values for PREPARE, DESCRIBE, FETCH, OPEN, CALL, or EXECUTE (continued)*

SQLTYPE	For PREPARE and DESCRIBE		For FETCH, OPEN, CALL, and EXECUTE	
	COLUMN DATA TYPE	SQLLEN	HOST VARIABLE DATA TYPE	SQLLEN
496/497	Large integer	4 (See Note #3)	Large integer	4
500/501	Small integer	2 (See Note #3)	Small integer	2
504/505	N/A	N/A	DISPLAY SIGN LEADING SEPARATE	Precision in byte 1; scale in byte 2
904/905	N/A	N/A	ROWID	40
916/917	N/A	N/A	BLOB file reference variable	267
920/921	N/A	N/A	CLOB file reference variable	267
924/925	N/A	N/A	DBCLOB file reference variable	267
960/961	N/A	N/A	BLOB locator	4
964/965	N/A	N/A	CLOB locator	4
968/969	N/A	N/A	DBCLOB locator	4

Notes:

1. The DataLink datatype is only returned on DESCRIBE TABLE.
2. The len.sqllonglen field in the secondary SQLVAR contains the length attribute of the column.
3. Large and small binary numbers can be represented in the SQL descriptor area (SQLDA) as either lengths 2 or 4. They can also be represented with the precision in byte 1 and the scale in byte 2. If the first byte is greater than X'00', it indicates precision and scale. Big integer numbers do not allow a precision and scale. The SQLDA defines them as length 8.

SQLVAR2

This is the Extended SQLVAR structure that contains 3 fields. Extended SQLVARs are needed for all columns of the result if the result includes any distinct type or LOB columns. For distinct types, they contain the distinct type name. For LOBs, they contain the length attribute of the host variable and a pointer to the buffer that contains the actual length. If locators are used to represent LOBs, these entries are not necessary. The number of Extended SQLVAR occurrences needed depends on the statement that the SQLDA was provided for and the data types of the columns or parameters being described. Byte 7 of SQLDAID is always set to the number of sets of SQLVARs necessary.

If SQLD is not set to a sufficient number of SQLVAR occurrences:

- SQLD is set to the total number of SQLVAR occurrences needed for all sets.
- A +237 warning is returned in the SQLCODE field of the SQLCA if at least enough were specified for the Base SQLVAR Entries. The Base SQLVAR entries are returned, but no Extended SQLVARs are returned.
- A +239 warning is returned in the SQLCODE field of the SQLCA if enough SQLVARs were not specified for even the Base SQLVAR Entries. No SQLVAR entries are returned.

SQLLONGLEN

SQLLONGLEN is a 4-byte integer variable that specifies the length attribute of a LOB (BLOB, CLOB, or DBCLOB) host variable or column.

SQLDATALEN

SQLDATALEN is a 16-byte pointer variable that specifies the address of the length of the host variable. This variable is used for LOB (BLOB, CLOB, and DBCLOB) host variables only. It is not used for DESCRIBE or PREPARE.

If this field is NULL, then the actual length of the data is stored in the 4 bytes immediately before the start of the data, and SQLDATA points to the first byte of the field length. The length indicates the number of bytes for a BLOB or CLOB, and the number of characters for a DBCLOB.

If this field is not NULL, it contains a pointer to a 4-byte long buffer that contains the actual length in bytes (even for DBCLOB) of the data in the buffer pointed to by the SQLDATA field in the matching base SQLVAR.

SQLDATATYPE_NAME

SQLDATATYPE_NAME is a variable-length character variable with a maximum length of 30. It is only used for DESCRIBE or PREPARE. This variable is set to one of the following:

- For a distinct type column, the database manager sets this to the fully qualified distinct type name. If the qualified name is longer than 30 bytes, it is truncated.
- For a label, the database manager sets this to the first 20 bytes of the label.
- For a column name, the database manager sets this to the column name.

Example: Select-statement for allocating storage for SQLDA

Suppose your application needs to be able to handle a dynamic SELECT statement; one that changes from one use to the next. This statement could be read from a display, passed in from another application, or built by your application on the fly. In other words, you don't know exactly what this statement is going to be returning every time. Your application needs to be able to handle the varying number of result columns with data types that are unknown ahead of time.

For example, the following statement needs to be processed:

```
SELECT WORKDEPT, PHONENO
FROM CORPDATA.EMPLOYEE
WHERE LASTNAME = 'PARKER'
```

Note: This SELECT statement has no INTO clause. Dynamic SELECT statements must *not* have an INTO clause, even if they return only one row.

The statement is assigned to a host variable. The host variable, in this case named DSTRING, is then processed by using the PREPARE statement as shown:

```
EXEC SQL
PREPARE S1 FROM :DSTRING;
```

Next, you need to determine the number of result columns and their data types. To do this, you need an SQLDA.

The first step in defining an SQLDA, is to allocate storage for it. (Allocating storage is not necessary in REXX.) The techniques for acquiring storage are language dependent. The SQLDA must be allocated on a 16-byte boundary. The SQLDA consists of a fixed-length header that is 16 bytes in length. The header is followed by a varying-length array section (SQLVAR), each element of which is 80 bytes in length.

The amount of storage that you need to allocate depends on how many elements you want to have in the SQLVAR array. Each column you select must have a corresponding SQLVAR array element. Therefore, the number of columns listed in your SELECT statement determines how many SQLVAR array elements you should allocate. Since this SELECT statement was specified at run time, it is impossible to know exactly how many columns will be accessed. Consequently, you must estimate the number of columns. Suppose, in this example, that no more than 20 columns are ever expected to be accessed by a single SELECT statement. In this case, the SQLVAR array should have a dimension of 20, ensuring that each item in the select-list has a corresponding entry in SQLVAR. This makes the total SQLDA size 20 x 80, or 1600, plus 16 for a total of 1616 bytes

Having allocated what you estimated to be enough space for your SQLDA, you need to set the SQLN field of the SQLDA equal to the number of SQLVAR array elements, in this case 20.

Having allocated storage and initialized the size, you can now issue a DESCRIBE statement.

```
EXEC SQL
DESCRIBE S1 INTO :SQLDA;
```

When the DESCRIBE statement is run, SQL places values in the SQLDA that provide information about the select-list for your statement. The following tables show the contents of the SQLDA after the DESCRIBE is run. Only the entries that are meaningful in this context are shown.

The SQLDA header would contain:

Table 30. SQLDA Header

Description	Value
SQLAID	'SQLDA'
SQLDABC	1616
SQLN	20
SQLD	2

SQLDAID is an identifier field initialized by SQL when a DESCRIBE is run. SQLDABC is the byte count or size of the SQLDA. The SQLDA header is followed by 2 occurrences of the SQLVAR structure, one for each column in the result table of the SELECT statement being described:

Table 31. SQLVAR Element 1

Description	Value
SQLTYPE	453
SQLLEN	3
SQLDATA (3:4)	37

Table 31. SQLVAR Element 1 (continued)

Description	Value
SQLNAME	8 WORKDEPT

Table 32. SQLVAR Element 2

Description	Value
SQLTYPE	453
SQLEN	4
SQLDATA(3:4)	37
SQLNAME	7 PHONENO

Your program might have to alter the SQLN value if the SQLDA is not large enough to contain the described SQLVAR elements. For example, suppose that instead of the estimated maximum of 20 columns, the SELECT statement actually returns 27. SQL cannot describe this select-list because the SQLVAR needs more elements than the allocated space allows. Instead, SQL sets the SQLD to the actual number of columns specified by the SELECT statement and the remainder of the structure is ignored. Therefore, after a DESCRIBE, you should compare the SQLN value to the SQLD value. If the value of SQLD is greater than the value of SQLN, allocate a larger SQLDA based on the value in SQLD, as follows, and perform the DESCRIBE again:

```
EXEC SQL
    DESCRIBE S1 INTO :SQLDA;
IF SQLN <= SQLD THEN
DO;

/*Allocate a larger SQLDA using the value of SQLD.*/
/*Reset SQLN to the larger value.*/

EXEC SQL
    DESCRIBE S1 INTO :SQLDA;
END;
```

If you use DESCRIBE on a non SELECT statement, SQL sets SQLD to 0. Therefore, if your program is designed to process both SELECT and non SELECT statements, you can describe each statement after it is prepared to determine whether it is a SELECT statement. This example is designed to process only SELECT statements; the SQLD value is not checked.

Your program must now analyze the elements of SQLVAR returned from the successful DESCRIBE. The first item in the select-list is WORKDEPT. In the SQLTYPE field, the DESCRIBE returns a value for the data type of the expression and whether nulls are applicable or not (see Table 29 on page 259).

In this example, SQL sets SQLTYPE to 453 in SQLVAR element 1. This specifies that WORKDEPT is a fixed-length character string result column and that nulls are permitted in the column.

SQL sets SQLEN to the length of the column. Because the data type of WORKDEPT is CHAR, SQL sets SQLEN equal to the length of the character column. For WORKDEPT, that length is 3. Therefore, when the SELECT statement is later run, a storage area large enough to hold a CHAR(3) string will be needed.

Because the data type of WORKDEPT is CHAR FOR SBCS DATA, the first 4 bytes of SQLDATA were set to the CCSID of the character column.

The last field in an SQLVAR element is a varying-length character string called SQLNAME. The first 2 bytes of SQLNAME contain the length of the character data. The character data itself is usually the name of a column used in the SELECT statement, in this case WORKDEPT. The exceptions to this are select-list items that are unnamed, such as functions (for example, SUM(SALARY)), expressions (for example, A+B-C), and constants. In these cases, SQLNAME is an empty string. SQLNAME can also contain a label rather than a name. One of the parameters associated with the PREPARE and DESCRIBE statements is the USING clause. You can specify it this way:

```
EXEC SQL
      DESCRIBE S1 INTO:SQLDA
      USING LABELS;
```

If you specify:

NAMES (or omit the USING parameter entirely)

Only column names are placed in the SQLNAME field.

SYSTEM NAMES

Only the system column names are placed in the SQLNAME field.

LABELS

Only labels associated with the columns listed in your SQL statement are entered here.

ANY Labels are placed in the SQLNAME field for those columns that have labels; otherwise, the column names are entered.

BOTH Names and labels are both placed in the field with their corresponding lengths. Remember to double the size of the SQLVAR array because you are including twice the number of elements.

ALL Column names, labels, and system column names are placed in the field with their corresponding lengths. Remember to triple the size of the SQLVAR array

For more information about the USING option, see the DESCRIBE statement and the SQLDA section in the *SQL Reference* book.

In this example, the second SQLVAR element contains the information for the second column used in the select: PHONENO. The 453 code in SQLTYPE specifies that PHONENO is a CHAR column. SQLLEN is set to 4.

Now you need to set up to use the SQLDA to retrieve values when running the SELECT statement.

After analyzing the result of the DESCRIBE, you can allocate storage for variables that are to contain the result of the SELECT statement. For WORKDEPT, a character field of length 3 must be allocated; for PHONENO, a character field of length 4 must be allocated. Since both of these results can be the NULL value, an indicator variable must be allocated for each field as well.

After the storage is allocated, you must set SQLDATA and SQLIND to point to the allocated storage areas. For each element of the SQLVAR array, SQLDATA points to the place where the result value is to be put. SQLIND points to the place where

the null indicator value is to be put. The following tables show what the structure looks like now. Only the entries that are meaningful in this context are shown:

Table 33. SQLDA Header

Description	Value
SQLAID	'SQLDA'
SQLDABC	1616
SQLN	20
SQLD	2

Table 34. SQLVAR Element 1

Description	Value
SQLTYPE	453
SQLLEN	3
SQLDATA	Pointer to area for CHAR(3) result
SQLIND	Pointer to 2 byte integer indicator for result column

Table 35. SQLVAR Element 2

Description	Value
SQLTYPE	453
SQLLEN	4
SQLDATA	Pointer to area for CHAR(4) result
SQLIND	Pointer to 2 byte integer indicator for result column

You are now ready to retrieve the SELECT statements results. Dynamically defined SELECT statements must not have an INTO statement. Therefore, all dynamically defined SELECT statements must use a cursor. Special forms of the DECLARE, OPEN, and FETCH are used for dynamically defined SELECT statements.

The DECLARE statement for the example statement is:

```
EXEC SQL DECLARE C1 CURSOR FOR S1;
```

As you can see, the only difference is that the name of the prepared SELECT statement (S1) is used instead of the SELECT statement itself. The actual retrieval of result rows is made as follows:

```
EXEC SQL
    OPEN C1;
EXEC SQL
    FETCH C1 USING DESCRIPTOR :SQLDA;
DO WHILE (SQLCODE = 0);
/*Process the results pointed to by SQLDATA*/
EXEC SQL
    FETCH C1 USING DESCRIPTOR :SQLDA;
END;
EXEC SQL
    CLOSE C1;
```

The cursor is opened. The result rows from the SELECT are then returned one at a time using a FETCH statement. On the FETCH statement, there is no list of output host variables. Instead, the FETCH statement tells SQL to return results into areas

described by your SQLDA. The results are returned into the storage areas pointed to by the SQLDATA and SQLIND fields of the SQLVAR elements. After the FETCH statement has been processed, the SQLDATA pointer for WORKDEPT has its referenced value set to 'E11'. Its corresponding indicator value is 0 since a non-null value was returned. The SQLDATA pointer for PHONENO has its referenced value set to '4502'. Its corresponding indicator value is also 0 since a non-null value was returned.

Parameter markers

In the example we used, the SELECT statement that was dynamically run had a constant value in the WHERE clause. In the example, it was:

```
WHERE LASTNAME = 'PARKER'
```

If you want to run the same SELECT statement several times, using different values for LASTNAME, you can use an SQL statement that would look like this:

```
SELECT WORKDEPT, PHONENO
FROM CORPDATA.EMPLOYEE
WHERE LASTNAME = ?
```

When your parameters are not predictable, your application cannot know the number or types of the parameters until run time. You can arrange to receive this information at the time your application is run, and by using a USING DESCRIPTOR on the OPEN statement, you can substitute the values contained in specific host variables for the parameter markers included in the WHERE clause of the SELECT statement.

To code such a program, you need to use the OPEN statement with the USING DESCRIPTOR clause. This SQL statement is used to not only open a cursor, but to replace each parameter marker with the value of the corresponding host variable. The descriptor name that you specify with this statement must identify an SQLDA that contains a valid description of those host variables. This SQLDA, unlike those previously described, is not used to return information on data items that are part of a SELECT list. That is, it is not used as output from a DESCRIBE statement, but as input to the OPEN statement. It provides information on host variables that are used to replace parameter markers in the WHERE clause of the SELECT statement. It gets this information from the application, which must be designed to place appropriate values into the necessary fields of the SQLDA. The SQLDA is then ready to be used as a source of information for SQL in the process of replacing parameter markers with host variable data.

When you use the SQLDA for input to the OPEN statement with the USING DESCRIPTOR clause, not all of its fields have to be filled in. Specifically, SQLDAID, SQLRES, and SQLNAME can be left blank (SQLNAME can be set if a specific CCSID is needed.) Therefore, when you use this method for replacing parameter markers with host variable values, you need to determine:

- How many parameter markers are there
- What are the data types and attributes of these parameters markers (SQLTYPE, SQLLEN, and SQLNAME)
- Do you want an indicator variable

In addition, if the routine is to handle both SELECT and non SELECT statements, you may want to determine what category of statement it is. (Alternatively, you can write code to look for the SELECT keyword.)

If your application uses parameter markers, your program has to:

1. Read a statement into the DSTRING varying-length character string host variable.
2. Determine the number of parameter markers.
3. Allocate an SQLDA of that size.
This is not applicable in REXX.
4. Set SQLN and SQLD to the number of ? parameter markers.
SQLN is not applicable in REXX.
5. Set SQLDABC equal to $SQLN * LENGTH(SQLVAR) + 16$.
This is not applicable in REXX.
6. For each parameter marker:
 - a. Determine the data types, lengths, and indicators.
 - b. Set SQLTYPE and SQLLEN.
 - c. Allocate storage to hold the input values (the ? values).
 - d. Set these values.
 - e. Set SQLDATA and SQLIND (if applicable) for each parameter marker.
 - f. If character variables are used, and they are in a CCSID other than the job default CCSID, set SQLNAME (SQLCCSID in REXX) accordingly.
 - g. If graphic variables are used and they have a CCSID other than the associated DBCS CCSID for the job CCSID, set the SQLNAME (SQLCCSID in REXX) to that CCSID.
 - h. Issue the OPEN statement with a USING DESCRIPTOR clause to open your cursor and substitute a host variable value for each of the parameter markers.

The statement can then be processed normally.

Chapter 15. Use of dynamic SQL through client interfaces

You can access DB2 UDB for iSeries data through client interfaces on the server. The following topics help you get started with required tasks:

- “Accessing data with Java”
- “Accessing data with Domino”
- “Accessing data with Open Database Connectivity (ODBC)”
- “Accessing data with Portable Application Solutions Environment (PASE)”

Accessing data with Java

You can access DB2 UDB for iSeries data in your Java programs with the Developer Kit for Java Database Connectivity (JDBC) driver. The driver lets you perform the following tasks.

- Access database files
- Access JDBC database functions with embedded Structured Query Language (SQL) for Java
- Run SQL statements and process results.

See the topic “Setting up to use the IBM Developer Kit for Java JDBC driver” in the iSeries Information Center for details on how you can use the JDBC driver.

Accessing data with Domino

Domino for iSeries is a Domino server product that lets you integrate data from DB2 UDB for iSeries databases and Domino databases in both directions. To take advantage of this integration, you need to understand and manage how authorizations work between the two types of databases. For details, see the Domino for iSeries category of the iSeries Information Center.

Accessing data with Open Database Connectivity (ODBC)

You use the iSeries Access for Windows ODBC Driver to enable your ODBC client applications to effectively share data with each other and with the server. See “ODBC administration” in the iSeries Access for Windows category of the iSeries Information Center.

You can find information about connecting using Linux in the iSeries ODBC Driver for Linux topic. This topic discusses installing Linux on an iSeries logical partition, and installing and using the iSeries ODBC Driver for Linux to access the iSeries database.

Accessing data with Portable Application Solutions Environment (PASE)

Portable Application Solutions Environment (PASE) is an integrated runtime environment for AIX (or other UNIX-like) applications running on the iSeries system. See “OS/400 PASE” in the Integrated operating environments category of the iSeries Information Center for more information.

Chapter 16. Advanced database functions using iSeries Navigator

This chapter covers some of the advanced functions for your database that can be performed using iSeries Navigator. Overview information and tips are provided. Chapter 3, “Getting started with iSeries Navigator Database” on page 33 contains some of the basic functions. The topics included here are:

- “Mapping your database using Database Navigator”
- “Querying your database using Run SQL Scripts” on page 274
- “Reconstructing SQL statements using Generate SQL” on page 277
- “Advanced table functions using iSeries Navigator” on page 277
- “Defining SQL objects using iSeries Navigator” on page 283
- “Creating an SQL Package” on page 284

For other functions that you can perform in iSeries Navigator that are not based on SQL, see the following topics in Database Programming:

- Display table, view, and index description
- Reorganizing a table
- Displaying locked rows

For performance related tasks, see the Database Performance and Query Optimization.

Mapping your database using Database Navigator

Database Navigator enables you to visually depict the relationships of database objects on your system. The visual depiction you create for your database is called a Database Navigator Map. In essence, the Database Navigator Map is a snapshot of your database and the relationships that exist between all of the objects in the map.

Using Database Navigator, you can explore the complex relationships of your database objects using a graphical representation that presents the tables in your database, the relationships between tables, and indexes and constraints that are attached to tables. After you connect to your system, you can use Database Navigator to:

- Create a Database Navigator map
- Add new objects to a map
- Change the objects to include in a map
- Create a user-defined relationship

The primary workspace for Database Navigator is a window that is divided into several main areas. These areas allow you to find the objects to include in a map, show and hide items in a map, view the map, and check on the status of changes pending for a map. The following provide a description of the main areas of the Database Navigator window.

Locator Pane

The Locator Pane, on the left side of the Database Navigator window, is used to find the objects that you want to include in your new map, or to locate

objects that are part of an open map. The upper Locator Pane is a search facility that can be used to specify the Name, Type, and Library of the objects that you want to include in the map. The results of the search are displayed in the lower Locator Pane under the Library Tree and Library Table tabs. When the results are displayed under these tabs, you can add objects to the map by right-clicking on an object and selecting Add to Map or double-clicking on the object name. Then, when the map is created, you can see a list of the objects in the map by clicking on the Objects In Map tab.

Map Pane

The Map Pane, on the right side of the Database Navigator window, graphically displays the database objects and their relationships. In the Map Pane you can:

- Add tables and views that exist on the system, but were not originally included in the current instance of the map
- Remove objects from the map
- Change object placement
- Zoom in or out on an object
- Make changes to objects in the map
- Generate the SQL for all objects in the map

Status Bars

Object Status Bar

The Object Status Bar, located on the bottom left of the Database Navigator window, displays the number of visible and eligible objects in the map.

Action Status Bar

The Action Status Bar, located on the bottom center of the Database Navigator window, provides a clear description of what has taken place in the map, and whether modifications are pending.

Modification Status Bar

The Modification Status Bar indicates whether a modification has been made or is pending.

Tips for using Database Navigator

- To change the size of either side of the window, drag the bar (splitter) that separates the two sides.
- Be sure to right-click the objects in both the left and right sides of the window. The right-click menus give you quick access to common functions.
- To quickly open a library and display the objects in it, double-click the library.
- To access the various Database Navigator commands use either the **Menu** bar or the **Toolbar**.

Creating a Database Navigator map

The visual depiction that you create of your database is called a Database Navigator Map. The map is actually a snapshot of your database data at the time you choose to create the map or refresh an existing map. This pictorial representation of a database is intended to provide you with the ability to understand existing complex databases, create a new database, communicate your database, and manage objects in your database. To create a Database Navigator Map:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Database Navigator**.

2. Right-click **Database Navigator** and select **New**.
3. In the **Database Navigator Locator Pane**, click the **Library tree** tab and then select the library that contains the tables, views, or indexes that you want to use to create the map.
4. Expand the object type (table, index, or view).
5. Right-click the object for which you want to create the map and select **Add to Map**, or double-click the object.

Note: You can also click the **Map your database** task on the task pad at the bottom of the **iSeries Navigator** window to create a map.

You can save this map by selecting **Exit** from the **File** menu. Then, if changes are pending, select **Yes** on the **Save Changes To** dialog. This map can be reopened at a later time.

Once you have created this map of your database, you can do the following:

- Add new objects to a map
- Change the objects to include in a map
- Create a user-defined relationship

Adding new objects to a map

With Database Navigator, you can create new SQL objects to add to your map. Among the objects that can be created are:

- Tables
- Journals
- Views

To create new SQL objects to be displayed in a map:

1. Create or open a Database Navigator map.
2. Right-click in the map pane and select **Create**.
3. Select the type of object that you want to create.

Changing the objects to include in a map

By default, Database Navigator searches for and includes all objects in your map. To limit the number of objects that are searched for, you can change the user preferences.

To change which objects to include in the map, do the following:

1. Create or open a Database Navigator Map.
2. From the **Options** menu, select **User Preferences**.
3. On the **User Preferences** dialog, in the **When adding an object to the map find these related objects** group box, select the objects you want to include, or deselect the objects you do not want to include.
4. Click **OK**.
5. If you want to refresh the map with the new preferences, click **Yes** in the Information box.

Creating a user-defined relationship

When you have relationships that are defined by your programs, you can create a user-defined relationship in Database Navigator so that your relationship is

displayed in the map. An example of this might be creating a user-defined relationship to remind programmers of an important join between two tables.

To add a user-defined relationship to your map, do the following:

1. Create or open a Database Navigator Map.
2. Right-click the map and select **Create**.
3. Select **User-Defined Relationship**.
4. Specify a **Name** and a **Description** for the user-defined relationship. Unlike some iSeries Navigator functions where the description is optional, it is important to provide a meaningful description for your user-defined relationship as it is the only way for you to indicate what the user-defined relationship represents.
5. Select the objects that you want to include in the relationship by selecting from the list of objects.
6. Choose the shape and color you want for the object.

Querying your database using Run SQL Scripts

The **Run SQL Scripts** window in iSeries Navigator allows you to create, edit, run, and troubleshoot scripts of SQL statements. When you have finished working with the scripts, they can be saved to your PC. You can use Run SQL Scripts to:

- Create an SQL script
- Run SQL scripts
- View the result set of a stored procedure
- View the job log
- Change the options for running an SQL script
- Generate SQL for database objects

The **Run SQL Scripts** window has several key areas:

Input

The upper portion of the **Run SQL Scripts** window is the **Input** pane. This area is used to create and edit the SQL statements that you want to run. You can create statements manually or select from the **Examples** list. You can also use the **Generate SQL** function to insert generated SQL at the current cursor position.

Examples

The **Examples** list box lists examples of SQL statements and Control Language (CL) commands. Select a statement and click Insert to place the example at the current cursor position in the Input pane.

Note: Each statement must be separated by a semicolon.

Output

The lower portion of the **Run SQL Scripts** window is the **Output** pane, which is comprised of the **Messages** tab and any additional tabs that display the output of the SQL statement that is run. The **Messages** tab provides feedback based on the execution of the SQL statement that is run.

Tips for the Run SQL Scripts window

- You can use the SQL scripts function without starting iSeries Navigator. Once you have saved a script file, you can double-click on the script file to use it without starting iSeries Navigator.

- Use a semicolon (;) to separate statements.
- Use two dashes (--) to make the rest of the line a comment.
- To make longer comments, start the comment with "/*" and use "*/" to end it. Comments made in this manner can be any length.
- Place the cursor in a statement and the statement is automatically selected when run.
- Control Language (CL) commands can be executed by placing CL: at the beginning of a statement. You can use any CL command that can be submitted from a batch job.

Creating an SQL script

To create an SQL script:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → .
2. Right-click on the database that you want to work with and select **Run SQL Scripts**.
3. From the **Run SQL Scripts** window, select **New**.
4. Manually create your statements, insert examples from the **Examples** list, or retrieve the SQL for an existing object using the generate SQL function.
5. After you have finished creating your statements, you can check the syntax of your statement by selecting **Syntax Check** from the **Run** menu.

When syntax checking is complete, you can save the script by selecting **Save** from the **File** menu. You are prompted for the location and name of the script.

Once you have created a script, you can:

- Run SQL scripts
- View the job log
- Change the options for running an SQL script
- Generate SQL for database objects

Running SQL scripts

To run an SQL script, select one of the following options from the **Run** menu:

- **All** - Runs your SQL script from the beginning to the end. If an error occurs and the **Stop on Error** option is turned on, the program stops and the statement where the error occurred remains selected.
- **From Selected** - Starts your SQL script from the first statement that is selected or from the current cursor position and continues to the end of the script.
- **Selected** - Runs the statements that are selected.

The results are added to the end of the **Messages** tab. If the **Smart Statement Selection** option on the **Options** menu is not checked, the text that is selected is run as a single SQL statement.

Changing the options for running an SQL script

To change the options for running an SQL script, select one of the following options from the **Options** menu:

Stop on Error

Turns stopping on errors on or off. If this option is selected and an error occurs, the SQL script stops running and the statement that resulted in an error remains selected.

Smart Statement Selection

Turns Smart Statement Selection on or off. When this option is selected, all highlighted statements are run in sequence when the **Selected** command on the **Run** menu is selected. If this option is not selected, the **Selected** command runs the highlighted text as a single SQL statement. Selecting **Smart Statement Selection** also ensures that complete statements are run even if one or more statements are only partially highlighted.

Display Results in Separate Window

Causes results from select statements to be displayed in a separate window instead of the **Output** pane.

Include Debug Messages in Job Log

Turns debugging for statements run in the **Run SQL Scripts** window on or off. You can see query optimizer and other database debugging messages by turning on this option, running your statements, and refreshing the **Job Log** window.

Viewing the result set for a stored procedure

You can view the result set for a stored procedure in Run SQL Scripts by typing in a CALL statement as an SQL statement. The results are available in the results window just like a normal query. If you have multiple results sets, they will each be represented by an additional results tab. Additionally, you can view the output parameters in the **Messages** tab. For more information about using CALL, see CALL in the SQL Reference book.

Viewing the Job Log

The **Job Log** displays messages related to your job. To see query optimizer and other database debugging messages, select **Include Debug Messages in Job Log** from the **Options** menu and run the statements again. If the **Job Log** dialog is open when you do this, refresh the view to see new messages. To view the Job Log:

From the **View** menu, select **Job Log**.

Note: The Job Log is not cleared when **Clear Run History** is used, so you can use the Job Log to see messages that are no longer in the **Output** pane. To stop or cancel an SQL scripts run, select one of the following options from the Run menu:

Stop After Current

Stops running the SQL script after the currently running statement ends.

Cancel Request

Requests that the system cancel the current SQL statement. However, as not all SQL statements can be canceled, the SQL statement may continue to completion even after this option is used. SQL statements that have already completed host processing before **Cancel Request** is pressed will also continue to completion. For example, Select statements that have already completed query processing but have not yet returned the results to the client usually cannot be canceled.

Reconstructing SQL statements using Generate SQL

Generate SQL allows you to reconstruct the SQL used to create existing database objects. This process is often referred to as reverse engineering. You can generate SQL for a Schema, Table, Type, View, Procedure, Function, Alias, and Index. Additionally, if you generate SQL for a table that has constraints or triggers associated with it, the SQL will be generated for those as well. You can generate the SQL for one object or many at a time. You also have the option of sending the generated SQL to Run SQL Scripts window for running or editing or you can write the generated SQL directly to a database or PC file.

For more information about using Generate SQL, see the following topics:

- Generate SQL for database objects
- Edit list of objects

Generate SQL for database objects

To generate the SQL used to create existing database objects:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. To generate SQL for a library, right-click on the library and select **Generate SQL**.
3. To generate SQL for an object contained within a library, click on the library that contains the object that you want to generate the SQL for.
4. Right-click the object you want to generate the SQL for and select **Generate SQL**.

You can change the default options for generating SQL by changing the values on the tabs. On the **Output** tab, you can choose a destination for the generated SQL. You can send the generated SQL to **Run SQL Scripts** or you can save it directly to a file, PC or system. On the **Options** tab, you can choose the **Standard** you want the generated SQL to conform to, and choose additionally, to include informational messages, include drop messages, generate labels, and format for readability. On the **Format** tab, you can select format options. These options must conform to the **Standards** options.

Editing list of objects for which to generate SQL

You can edit the list of objects for which to generate SQL. To add an object:

1. Click **Add**.
2. On the **Edit list of objects** dialog, navigate to the object that you want to include and select **Add**.
3. Click **OK** to return to the main generate SQL dialog.

To remove an object from the list:

1. Select the object that you want to remove from the **Objects for which to generate SQL**.
2. Click **Remove**.

Advanced table functions using iSeries Navigator

Chapter 3, “Getting started with iSeries Navigator Database” on page 33 explains some of the basic table functions that you can perform. However, you can also use iSeries Navigator to:

- Create an alias

- Add an index
- Add a key constraint
- Add a check constraint
- Add a referential constraint
- Add a trigger
- Enable and disable a trigger
- Remove a constraint or trigger

Creating an alias using iSeries Navigator

An alias is an alternate name for a table or view. You can use an alias to refer to a table or view in those cases where an existing table or view can be referred to. You can create an alias for a table or view or for a member of a table or view.

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Right-click the library that you want to create the new alias in.
3. From the pop-up menu, select **New**, then **Alias**.
4. In the **Alias** field of the **New Alias** dialog, specify a name for the alias you are creating. The name cannot be the same as any index, table, view, file, or alias that already exists on the server.
5. In the **Description** field, specify a description of the new alias. This description can be up to 50 characters long. This field is optional.
6. In the **Table/View** field, specify the table or view that you want the alias to point to.
7. In the **Library** field, specify the library that contains the table or view you want the alias to point to.
8. Click **Advanced**.
9. On the **Advanced** dialog, if you want to create an alias for a table or a view, click **Create alias for a table or view**. This is the default option.
10. To create an alias for a member of a table or view, click **Create alias for a member of a table or view**. In the combo box, either enter or select the member you want the alias to point to.
11. Click **OK** to return to the **New Alias** dialog.
12. Click **OK** to create the alias.


Note: You can also create an alias by right-clicking on a table or view and selecting **Create Alias**.

For more information about aliases, see “Creating and using ALIAS names” on page 55.

Adding indexes using iSeries Navigator

You can use indexes to sort and select data. In addition, indexes help the system retrieve data faster for better query performance.

You can create a number of indexes. However, because the indexes are maintained by the system, a large number of indexes can adversely affect performance. For more information about indexes and query performance, see Effectively Using SQL Indexes in the *Database Performance and Query Optimization* information.

One type of index, the encoded vector index, allows for faster scans that can be more easily processed in parallel. For more information about accelerating your queries with encoded vector indexes , go to the DB2 for iSeries webpages.

You can create an index on a new or existing table. You can create a radix index or a encoded vector index using iSeries Navigator:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Click the library that contains the table in which you want to add an index.
3. In the detail pane, right-click the table in which you want to add an index and select **Properties**.
4. On the **Table Properties** or **New Table** dialog, select the **Indexes** tab.
5. On the **Indexes** tab, click **New**.
6. In the **Index** field, specify a name for the new index.
7. In the **Library** field, select the library where the index will reside.
8. Select which columns of the table will make up the index. To add a column, click on it and a number appears on the left. The number determines the key position of the column in the index. To remove a column from the index, click on it again.
9. To change the order of a key field from ascending sequence to descending sequence (or descending to ascending sequence), click the second column.
10. Select an **Index type**.
11. Select the number of distinct values if you are creating an encoded vector index.
12. Click **OK** to create the index.

Note: You can also add an index to a new table on the **New Table in** dialog.

You may modify a constraint only if it has been defined during your current table editing session. If you added the constraint and then clicked **OK** on either the **New Table** dialog or **Table Properties** dialog, then you have read only access for the constraint. If you want to change the constraint properties, you must drop the constraint and then recreate it with the appropriate changes.

For more information about creating indexes, see “Adding indexes” on page 57.

Adding key constraints using iSeries Navigator

Constraints are rules enforced by the database manager. DB2 UDB for iSeries supports two types of key constraints:

- A unique key constraint is the rule that the values of the key are valid only if they are unique. Unique constraints are enforced during the execution of **INSERT** and **UPDATE** statements.
- A primary key constraint is a form of unique constraint. The difference is that a primary key cannot contain any nullable columns.

To create a key constraint, do the following:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Double-click the library that contains the table to which you want to add the key constraint.

3. Right-click the table to which you want to add the key and select **Properties**.
4. On the **Table Properties** dialog, select the **Key Constraints** tab.
5. On the **Key Constraints** tab, click **New**.
6. On the **New Key Constraint** dialog, specify a name in the name text box. If no name is specified, the system automatically generates a name.
7. Select the column to which you want to add the key.
8. Select **Primary** to create a primary key or **Unique** to create a unique key.
9. Click **OK** to return to the **Table Properties** dialog.
10. Click **OK** to create the key.

Note: You can also add a key constraint to a new table on the **New Table in** dialog.

You may modify a constraint only if it has been defined during your current table editing session. If you added the constraint and then clicked **OK** on either the **New Table** dialog or **Table Properties** dialog, then you have read only access for the constraint. If you want to change the constraint properties, you must drop the constraint and then recreate it with the appropriate changes.

For more information about adding key constraints, see Chapter 10, “Data Integrity” on page 135.

Adding check constraints using iSeries Navigator

A check constraint assures the validity of data during inserts and updates by limiting the allowable values in a column or group of columns.

To create a check constraint, do the following:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Click the library that contains the table to which you want to add the check constraint.
3. Right-click the table to which you want to add the check constraint and select **Properties**.
4. On the **Table Properties** dialog, click the **Check Constraints** tab.
5. On the **Check Constraints** tab, click **New**.
6. On the **Check Constraint Search Condition** dialog, specify a name in the name text box. If no name is specified, the system automatically generates a name.
7. From the **Columns** list, double-click the column that you want to add the constraint to. The column appears in the clause box.
8. To insert an operator from the list, double-click it. The operator appears in the clause box.
9. To insert a function from the list, double-click it. You can modify the list by selecting the function type from the drop-down list. Once you have double-clicked the function, it will appear in the clause box.
10. Modify the expression until it is correct.
11. Click **OK** to return to the **Table Properties** dialog.
12. Click **OK** to create the check constraint.

Note: You can also add a check constraint to a new table on the **New Table in** dialog.

You may modify a constraint only if it has been defined during your current table editing session. If you added the constraint and then clicked **OK** on either the **New Table** dialog or **Table Properties** dialog, then you have read only access for the constraint. If you want to change the constraint properties, you must drop the constraint and then recreate it with the appropriate changes.

For more information about adding check constraints, see “Adding and using check constraints” on page 135.

Adding referential constraints using iSeries Navigator

Referential integrity is the condition of a set of tables in a database in which all references from one table to another are valid. You can ensure referential integrity in your database by adding referential constraints.

To add a referential constraint, do the following:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Double-click the library that contains the dependent table.
3. Right-click the dependent table and select **Properties**.
4. On the **Table Properties** dialog, select the **Referential Constraints** tab.
5. On the **Referential Constraints** tab, click **New**.
6. On the **New Referential Constraint** dialog, specify a name for the constraint. If no name is specified, the system automatically generates a name.
7. Select the column that you want to be dependent on the parent key value in the parent table.
8. Select the library that contains the parent table.
9. Select the table that contains the parent key.
10. Select the parent key to reference.
11. Select the delete action.
12. Select the update action (The action for insert is default).
13. Click **OK** to return to the **Table Properties** dialog.
14. Click **OK** to create the referential constraint

Note: You can also add a referential constraint to a new table on the **New Table in** dialog.

You may modify a constraint only if it has been defined during your current table editing session. If you added the constraint and then clicked **OK** on either the **New Table** dialog or **Table Properties** dialog, then you have read only access for the constraint. If you want to change the constraint properties, you must drop the constraint and then recreate it with the appropriate changes.

For more information about adding referential constraints, see “Adding or dropping referential constraints” on page 137.

Adding triggers using iSeries Navigator

A trigger is a set of actions that are run automatically when a specified change operation is performed on a specified physical database file. In this discussion, a table is a physical file. The change operation can be an insert, update, or delete

high level language statement in an application program, or an SQL INSERT, UPDATE, or DELETE statement. Triggers are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail.

Using iSeries Navigator, you can define system triggers and SQL triggers. Additionally, you can enable or disable a trigger.

To add a trigger, do the following:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Click the library that contains the table to which you want to add the trigger.
3. Right-click the table to which you want to add the trigger and select **Properties**. On the **Table Properties** dialog, click the **Triggers** tab.
4. Select **Add system trigger** to add a system trigger.
5. Select **Add SQL trigger** to add an SQL trigger.

For more information about system triggers, see Triggering automatic events in your database in the *Database Programming* book.

For more information about SQL triggers, see “SQL triggers” on page 148.

Enabling and disabling a trigger

Triggers need to be enabled in order to run. However, disabling a trigger allows you to work with the table without causing the trigger to run.

To enable/disable a trigger, do the following:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Click the library that contains the table that you want to enable/disable a trigger for.
3. Right-click the table and select **Properties**.
4. On the **Table Properties** dialog, click the **Trigger** tab.
5. Select the trigger that you want to enable/disable and click **Enable** to enable the trigger or **Disable** to disable the trigger.

For information on using the CHGPFTRG CL command to enable and disable a trigger, see Enabling and disabling a trigger in the *Database Programming* book.

Removing constraints and triggers

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Click the library that contains the table from which you want to remove the constraint or trigger.
3. Right-click the table from which you want to remove the constraint or trigger and select **Properties**.
4. On the **Table Properties** dialog, click the tab for the type of constraint or trigger that you want to remove.
5. Select the constraint that you want to remove and click **Delete**.

Defining SQL objects using iSeries Navigator

iSeries Navigator provides you with a simple means of defining some SQL objects on the system. For example, you can define:

- **Procedures:** A procedure (often called a stored procedure) is a program that can be called to perform operations that can include both host language statements and SQL statements. Procedures in SQL provide the same benefits as procedures in a host language. That is, a common piece of code need only be written and maintained once and can be called from several programs. For more information about procedures, see Chapter 11, “Stored Procedures” on page 159.
- **User-defined functions:** User-defined functions (UDFs) consist of three types: sourced, external, and SQL. Sourced function UDFs call other functions to perform the operation. SQL and external function UDFs require that you write and execute separate code. You can create scalar, column, and table functions. For more information about Functions, see “User-defined functions (UDF)” on page 200.
- **User-defined types:** A user-defined distinct type is a mechanism that allows you to extend DB2 capabilities beyond the built-in data types available. User-defined distinct types enable you to define new data types to DB2 which gives you considerable power since you are no longer restricted to using the system-supplied built-in data types to model your business and capture the semantics of your data. Distinct data types allow you to map on a one-to-one basis to existing database types. For more information about types, see “User-defined distinct types (UDT)” on page 215.

For more information about defining these objects using iSeries Navigator, see the following topics:

- “Defining a stored procedure using iSeries Navigator”
- “Defining a user-defined function using iSeries Navigator”
- “Defining a user-defined type using iSeries Navigator” on page 284

Defining a stored procedure using iSeries Navigator

If you want to call a program as a procedure from an SQL program, you must first define the program as an external procedure. The program for which a procedure is defined does not need to exist at the time the procedure is defined.

To define a program as a procedure, do the following:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Right-click on the library in which you want to define the function and select **New**.
3. Select **Procedure**.
4. Select **External** to create an external stored procedure.
5. Select **SQL** to create an SQL stored procedure.

For more information about creating and defining external stored procedures, see “Defining an external procedure” on page 160.

For more information about creating and defining SQL stored procedures, see “Defining an SQL procedure” on page 161.

Defining a user-defined function using iSeries Navigator

To define a program as a user-defined function, do the following

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Right-click on the library in which you want to define the function and select **New**.
3. Select **Function**.
4. Select **External** to create an external user-defined function.
5. Select **SQL** to create an SQL user-defined function.
6. Select **Sourced** to create a function based on another function.

For more information about creating and defining external functions, see “User-defined functions (UDF)” on page 200.

Defining a user-defined type using iSeries Navigator

Creating a new user-defined data type based on an existing data type can give you greater control over your data.

To create a user-defined type, do the following

1. In the **iSeries Navigator** window, expand your Server → **Databases** → the database that you want to work with → **Libraries**.
2. Right-click on the library in which you want to define the type and select **New**.
3. Select **Type**.
4. On the **New Type** dialog, specify the name that you want to give the new type in the **Type** field.
5. Specify the data type on which the new type is based in the **Type** field, in the **Source data type** section.
6. Click **OK**.

For more information about creating and defining user-defined types, see “User-defined distinct types (UDT)” on page 215.

Creating an SQL Package

SQL packages are permanent objects used to store information related to prepared SQL statements. They are used by ODBC support when the “extended dynamic” box is checked on a data source.

To create an SQL package:

1. In the **iSeries Navigator** window, expand your Server → **Databases**
2. Right click the database you want to use, and select **New SQL package**.
3. On the **Create SQL package** dialog, specify parameters as necessary.
4. Click **OK**.

For a complete listing of the parameters, see “CRTSQLPKG (Create Structured Query Language Package) Command” on page 375.

Chapter 17. Using Interactive SQL

This chapter describes how to use interactive SQL to run SQL statements and use the prompt function. Overview information and tips on using interactive SQL are provided. If you want to learn how to use SQL, you should see Chapter 2, "Getting Started with SQL". Special considerations for using interactive SQL with a remote connection are covered in "Accessing remote databases with interactive SQL" on page 295.

For more details, see "Basic functions of interactive SQL".

Notes:

1. The term *collection* is used synonymously with *schema*.
2. See "Code disclaimer information" on page x information for information pertaining to code examples.

Basic functions of interactive SQL

Interactive SQL allows the programmer or database administrator to quickly and easily define, update, delete, or look at data for testing, problem analysis, and database maintenance. A programmer, using interactive SQL, can insert rows into a table and test the SQL statements before running them in an application program. A database administrator can use interactive SQL to grant or revoke privileges, create or drop schemas, tables, or views, or select information from system catalog tables.

After an interactive SQL statement is run, a completion message or an error message is displayed. In addition, status messages are normally displayed during long-running statements.

You can see help on a message by positioning the cursor on the message and pressing F1=Help.

The basic functions supplied by interactive SQL are:

- The **statement entry** function allows you to:
 - Type in an interactive SQL statement and run it.
 - Retrieve and edit statements.
 - Prompt for SQL statements.
 - Page through previous statements and messages.
 - Call session services.
 - Invoke list selection function.
 - Exit interactive SQL.
- The **prompt** function allows you to type either a complete SQL statement or a partial SQL statement, press F4=Prompt, and then be prompted for the syntax of the statement. It also allows you to press F4 to get a menu of all SQL statements. From this menu, you can select a statement and be prompted for the syntax of the statement.
- The **list selection function** allows you to select from lists of your authorized relational databases, schemas, tables, views, columns, constraints, or SQL packages.

The selections you make from the lists can be inserted into the SQL statement at the cursor position.

- The **session services** function allows you to:
 - Change session attributes.
 - Print the current session.
 - Remove all entries from the current session.
 - Save the session in a source file.

For more details, see the following sections:

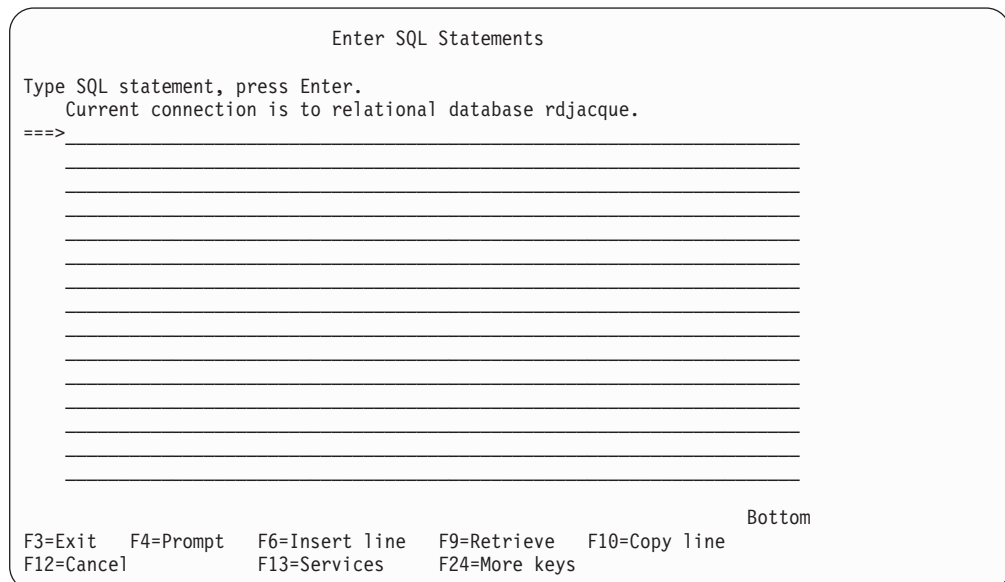
- “Starting interactive SQL”
- “Using statement entry function” on page 287
- “Prompting” on page 288
- “Using the list selection function” on page 290
- “Session services description” on page 293
- “Exiting interactive SQL” on page 294
- “Using an existing SQL session” on page 295
- “Recovering an SQL session” on page 295
- “Accessing remote databases with interactive SQL” on page 295

Starting interactive SQL

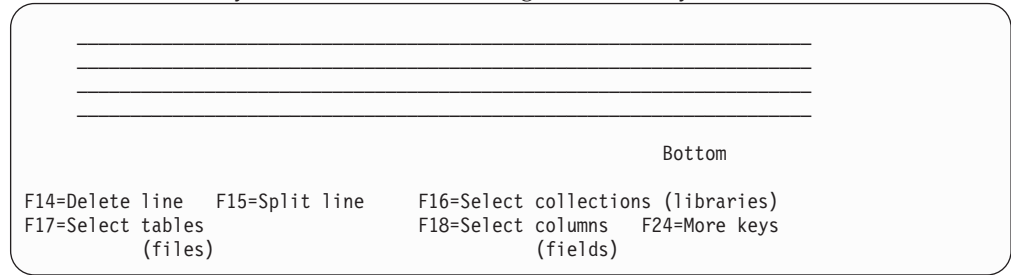
You can start using interactive SQL by typing STRSQL on an OS/400 command line. For a complete description of the command and its parameters, see Appendix B, “DB2 UDB for iSeries CL Command Descriptions”.

The Enter SQL Statements display appears. This is the main interactive SQL display. From this display, you can enter SQL statements and use:

- F4=prompt
- F13=Session services
- F16=Select collections
- F17=Select tables
- F18=Select columns



Press F24=More keys to view the remaining function keys.



Note: If you are using the system naming convention, the names in parentheses appear instead of the names shown above.

An interactive session consists of:

- Parameter values you specified for the STRSQL command.
- SQL statements you entered in the session along with corresponding messages that follow each SQL statement
- Values of any parameters you changed using the session services function
- List selections you have made

Interactive SQL supplies a unique session-ID consisting of your user ID and the current work station ID. This session-ID concept allows multiple users with the same user ID to use interactive SQL from more than one work station at the same time. Also, more than one interactive SQL session can be run from the same work station at the same time from the same user ID.

If an SQL session exists and is being re-entered, any parameters specified on the STRSQL command are ignored. The parameters from the existing SQL session are used.

Using statement entry function

The statement entry function is the function you first enter when selecting interactive SQL. You return to the statement entry function after processing each interactive SQL statement.

In the statement entry function, you type or prompt for the entire SQL statement and then submit it for processing by pressing the Enter key.

Typing statements

The statement you type on the command line can be one or more lines long. You cannot type comments for the SQL statement in interactive SQL. When the statement has been processed, the statement and the resulting message are moved upward on the display. You can then enter another statement.

If a statement is recognized by SQL but contains a syntax error, the statement and the resulting text message (syntax error) are moved upward on the display. In the input area, a copy of the statement is shown with the cursor positioned at the syntax error. You can place the cursor on the message and press F1=Help for more information about the error.

You can page through previous statements, commands, and messages. Press F9=Retrieve with your cursor on a previous statement to place a copy of that statement in the input area. If you need more room to type an SQL statement, page down on the display.

Prompting

The prompt function helps you supply the necessary information for the syntax of the statement you want to use. The prompt function can be used in any of the three statement processing modes: *RUN, *VLD, and *SYN.

You have two options when using the prompter:

- Type the verb of the statement before pressing F4=Prompt.

The statement is parsed and the clauses that are completed are filled in on the prompt displays.

If you type SELECT and press F4=Prompt, the following display appears:

```

Specify SELECT Statement

Type SELECT statement information.  Press F4 for a list.

FROM tables . . . . . _____
SELECT columns . . . . . _____
WHERE conditions . . . . . _____
GROUP BY columns . . . . . _____
HAVING conditions . . . . . _____
ORDER BY columns . . . . . _____
FOR UPDATE OF columns . . . . . _____

Bottom

Type choices, press Enter.

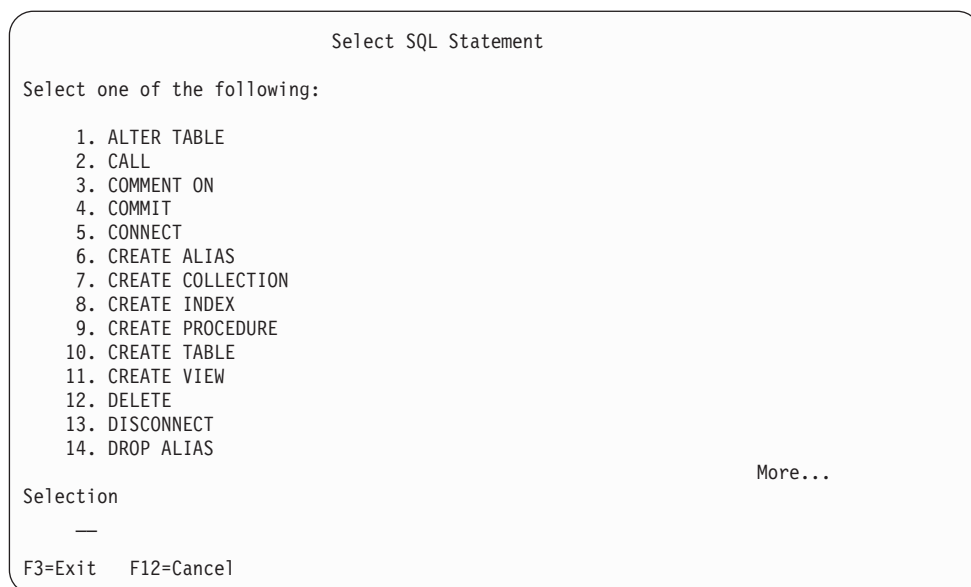
DISTINCT rows in result table . . . . . N  Y=Yes, N=No
UNION with another SELECT . . . . . N  Y=Yes, N=No
Specify additional options . . . . . N  Y=Yes, N=No

F3=Exit      F4=Prompt  F5=Refresh  F6=Insert line  F9=Specify subquery
F10=Copy line  F12=Cancel  F14=Delete line  F15=Split line  F24=More keys

```

- Press F4=Prompt before typing anything on the Enter SQL Statements display. You are shown a list of statements. The list of statements varies and depends on the current interactive SQL statement processing mode. For syntax check mode with a language other than *NONE, the list includes all SQL statements. For run and validate modes, only statements that can be run in interactive SQL are shown. You can select the number of the statement you want to use. The system prompts you for the statement you selected.

If you press F4=Prompt without typing anything, the following display appears:



If you press F21=Display Statement on a prompt display, the prompter displays the formatted SQL statement as it was filled in to that point.

When Enter is pressed within prompting, the statement that was built through the prompt screens is inserted into the session. If the statement processing mode is *RUN, the statement is run. The prompter remains in control if an error is encountered.

Syntax checking

The syntax of the SQL statement is checked when it enters the prompter. The prompter does not accept a syntactically incorrect statement. You must correct the syntax or remove the incorrect part of the statement or prompting will not be allowed.

Statement processing mode

The statement processing mode can be selected on the Change Session Attributes display. In *RUN (run) or *VLD (validate) mode, only statements that are allowed to run in interactive SQL can be prompted. In *SYN (syntax check) mode, all SQL statements are allowed. The statement is not actually run in *SYN or *VLD modes; only the syntax and existence of objects are checked.

Subqueries

Subqueries can be selected on any display that has a WHERE or HAVING clause. To see the subquery display, press F9=Specify subquery when the cursor is on a WHERE or HAVING input line. A display appears that allows you to type in subselect information. If the cursor is within the parentheses of the subquery when F9 is pressed, the subquery information is filled in on the next display. If the cursor is outside the parentheses of the subquery, the next display is blank. For more information about subqueries, see “Subqueries in SELECT statements” on page 105.

CREATE TABLE prompting

When prompting for CREATE TABLE, support is available for entering column definitions individually. Place your cursor in the column definition section of the display, and press F4=Prompt. A display that provides room for entering all the information for one column definition is shown.

To enter a column name longer than 18 characters, press F20=Display entire name. A window with enough space for a 30 character name will be displayed.

The editing keys, F6=Insert line, F10=Copy line, and F14=Delete line, can be used to add and delete entries in the column definition list.

Entering DBCS Data

The rules for processing DBCS data across multiple lines are the same on the Enter SQL Statements display and in the SQL prompter. Each line must contain the same number of shift-in and shift-out characters. When processing a DBCS data string that requires more than one line for entering, the extra shift-in and shift-out characters are removed. If the last column on a line contains a shift-in and the first column of the next line contains a shift-out, the shift-in and shift-out characters are removed by the prompter when the two lines are assembled. If the last two columns of a line contain a shift-in followed by a single-byte blank and the first column of the next line contains a shift-out, the shift-in, blank, shift-out sequence is removed when the two lines are assembled. This removal allows DBCS information to be read as one continuous character string.

As an example, suppose the following WHERE condition were entered. The shift characters are shown here at the beginning and end of the string sections on each of the two lines.

Specify SELECT Statement

Type SELECT statement information. Press F4 for a list.

FROM tables	TABLE1 _____
SELECT columns	* _____
WHERE conditions	COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQ> <RRSS>' _____
GROUP BY columns	_____
HAVING conditions	_____
ORDER BY columns	_____
FOR UPDATE OF columns	_____

When Enter is pressed, the character string is put together, removing the extra shift characters. The statement would look like this on the Enter SQL Statements display:

```
SELECT * FROM TABLE1 WHERE COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSS>'
```

Using the list selection function

The list selection function is available by pressing F4 on certain prompt displays, or F16, F17, or F18 on the Enter SQL Statements display. After pressing the function key, you are given a list of authorized relational databases, schemas, tables, views, aliases, columns, constraints, procedures, parameters, or packages from which to choose. If you request a list of tables, but you have not previously selected a schema, you are asked to select a schema first.

On a list, you can select one or more items, numerically specifying the order in which you want them to appear in the statement. When the list function is exited, the selections you made are inserted at the position of the cursor on the display you came from.

Always select the list you are primarily interested in. For example, if you want a list of columns, but you believe that the columns you want are in a table not currently selected, press F18=Select columns. Then, from the column list, press F17

to change the table. If the table list were selected first, the table name would be inserted into your statement. You would not have a choice for selecting columns.

You can request a list at any time while typing an SQL statement on the Enter SQL Statements display. The selections you make from the lists are inserted on the Enter SQL Statements display. They are inserted where the cursor is located in the numeric order that you specified on the list display. Although the selected list information is added for you, you must type the keywords for the statement.

The list function tries to provide qualifications that are necessary for the selected columns, tables, and SQL packages. However, sometimes the list function cannot determine the intent of the SQL statement. You need to review the SQL statement and verify that the selected columns, tables, and SQL packages are properly qualified.

Example: Using the list selection function

The following example shows you how to use the list function to build a SELECT statement.

Assume you have:

- Just entered interactive SQL by typing STRSQL on an OS/400 command line.
- Made no list selections or entries.
- Selected *SQL for the naming convention.

Note: The example shows lists that are not on your server. They are used as an example only.

Begin using SQL statements:

1. Type SELECT on the first statement entry line.
2. Type FROM on the second statement entry line.
3. Leave the cursor positioned after FROM.

```
Enter SQL Statements

Type SQL statement, press Enter.
===> SELECT
      FROM _
```

4. Press F17=Select tables to obtain a list of tables, because you want the table name to follow FROM.

Instead of a list of tables appearing as you expected, a list of collections appears (the Select and Sequence collections display). You have just entered the SQL session and have not selected a schema to work with.

5. Type a 1 in the *Seq* column next to YOURCOLL2 schema.

Select and Sequence Collections

Type sequence numbers (1-999) to select collection, press Enter.

Seq	Collection	Type	Text
	YOURCOLL1	SYS	Company benefits
1	YOURCOLL2	SYS	Employee personal data
	YOURCOLL3	SYS	Job classifications/requirements
	YOURCOLL4	SYS	Company insurances

6. Press Enter.

The Select and Sequence Tables display appears, showing the tables existing in the YOURCOLL2 schema.

7. Type a 1 in the *Seq* column next to PEOPLE table.

Select and Sequence Tables

Type sequence numbers (1-999) to select tables, press Enter.

Seq	Table	Collection	Type	Text
	EMPLCO	YOURCOLL2	TAB	Employee company data
1	PEOPLE	YOURCOLL2	TAB	Employee personal data
	EMPLEXP	YOURCOLL2	TAB	Employee experience
	EMPLEVL	YOURCOLL2	TAB	Employee evaluation reports
	EMPLBEN	YOURCOLL2	TAB	Employee benefits record
	EMPLMED	YOURCOLL2	TAB	Employee medical record
	EMPLINVST	YOURCOLL2	TAB	Employee investments record

8. Press Enter.

The Enter SQL Statements display appears again with the table name, YOURCOLL2.PEOPLE, inserted after FROM. The table name is qualified by the schema name in the *SQL naming convention.

Enter SQL Statements

Type SQL statement, press Enter.

```
====> SELECT
      FROM YOURCOLL2.PEOPLE _
```

9. Position the cursor after SELECT.

10. Press F18=Select columns to obtain a list of columns, because you want the column name to follow SELECT.

The Select and Sequence Columns display appears, showing the columns in the PEOPLE table.

11. Type a 2 in the *Seq* column next to the NAME column.
12. Type a 1 in the *Seq* column next to the SOCSEC column.

Select and Sequence Columns					
Type sequence numbers (1-999) to select columns, press Enter.					
Seq	Column	Table	Type	Digits	Length
2	NAME	PEOPLE	CHARACTER		6
	EMPLNO	PEOPLE	CHARACTER		30
1	SOCSEC	PEOPLE	CHARACTER		11
	STRADDR	PEOPLE	CHARACTER		30
	CITY	PEOPLE	CHARACTER		20
	ZIP	PEOPLE	CHARACTER		9
	PHONE	PEOPLE	CHARACTER		20

13. Press Enter.

The Enter SQL Statements display appears again with SOCSEC, NAME appearing after SELECT.

Enter SQL Statements	
Type SQL statement, press Enter.	
===>	SELECT SOCSEC, NAME FROM YOURCOLL2.PEOPLE

14. Press Enter.

The statement you created is now run.

Once you have used the list function, the values you selected remain in effect until you change them or until you change the list of schemas on the Change Session Attributes display.

Session services description

The interactive SQL Session Services display is requested by pressing F13 on the Enter SQL Statements display.

From this display you can change session attributes and print, clear, or save the session to a source file.

Option 1 (Change session attributes) displays the Change Session Attributes display, which allows you to select the current values that are in effect for your interactive SQL session. The options shown on this display change based on the statement processing option selected.

The following session attributes can be changed:

- Commitment control attributes.
- The statement processing control.
- The SELECT output device.
- The list of schemas.
- The list type to select either all your system and SQL objects, or only your SQL objects.
- The data refresh option when displaying data.
- The allow copy data option.
- The naming option.
- The programming language.

- The date format.
- The time format.
- The date separator.
- The time separator.
- The decimal point representation.
- The SQL string delimiter.
- The sort sequence.
- The language identifier.

Option 2 (Print current session) accesses the Change Printer display, which lets you print the current session immediately and then continue working. You are prompted for printer information. All the SQL statements you entered and all the messages displayed are printed just as they appear on the Enter SQL Statements display.

Option 3 (Remove all entries from current session) lets you remove all the SQL statements and messages from the Enter SQL Statements display and the session history. You are prompted to ensure that you really want to delete the information.

Option 4 (Save session in source file) accesses the Change Source File display, which lets you save the session in a source file. You are prompted for the source file name. This function lets you embed the source file into a host language program by using the source entry utility (SEU).

Note: Option 4 allows you to embed prototyped SQL statements in a high-level language (HLL) program that uses SQL. The source file created by option 4 may be edited and used as the input source file for the Run SQL Statements (RUNSQLSTM) command.

Exiting interactive SQL

Pressing F3=Exit on the Enter SQL Statements display allows you to exit the interactive SQL environment and do one of the following:

1. Save and exit session. Leave interactive SQL. Your current session will be saved and used the next time you start interactive SQL.
2. Exit without saving session. Leave interactive SQL without saving your session.
3. Resume session. Remain in interactive SQL and return to the Enter SQL Statements display. The current session parameters remain in effect.
4. Save session in source file. Save the current session in a source file. The Change Source File display is shown to allow you to select where to save the session. You cannot recover and work with this session again in interactive SQL.

Notes:

1. Option 4 allows you to embed prototype SQL statements in a high-level language (HLL) program that uses SQL. Use the source entry utility (SEU) to copy the statements into your program. The source file can also be edited and used as the input source file for the Run SQL Statements (RUNSQLSTM) command.
2. If rows have been changed and locks are currently being held for this unit of work and you attempt to exit interactive SQL, a warning message is displayed.

Using an existing SQL session

If you saved only one interactive SQL session by using option 1 (Save and exit session) on the Exit Interactive SQL display, you may resume that session at any workstation. However, if you use option 1 to save two or more sessions on different workstations, interactive SQL will first attempt to resume a session that matches your work station. If no matching sessions are available, then interactive SQL will increase the scope of the search to include all sessions that belong to your user ID. If no sessions for your user ID are available, the system will create a new session for your user ID and current workstation.

For example, you saved a session on workstation 1 and saved another session on workstation 2 and you are currently working at workstation 1. Interactive SQL will first attempt to resume the session saved for workstation 1. If that session is currently in use, interactive SQL will then attempt to resume the session that was saved for workstation 2. If that session is also in use, then the system will create a second session for workstation 1.

However, suppose you are working at workstation 3 and want to use the ISQL session associated with workstation 2. You then may need to first delete the session from workstation 1 by using option 2 (Exit without saving session) on the Exit Interactive SQL display.

Recovering an SQL session

If the previous SQL session ended abnormally, interactive SQL presents the Recover SQL Session display at the start of the next session (when the next STRSQL command is entered). From this display, you can either:

- Recover the old session by selecting option 1 (Attempt to resume existing SQL session).
- Delete the old session and start a new session by selecting option 2 (Delete existing SQL session and start a new session).

If you choose to delete the old session and continue with the new session, the parameters you specified when you entered STRSQL are used. If you choose to recover the old session, or are entering a previously saved session, the parameters you specified when you entered STRSQL are ignored and the parameters from the old session are used. A message is returned to indicate which parameters were changed from the specified value to the old session value.

Accessing remote databases with interactive SQL

In interactive SQL, you can communicate with a remote relational database by using the SQL CONNECT statement. Interactive SQL uses the CONNECT (Type 2) semantics (distributed unit of work) for CONNECT statements. Interactive SQL does an implicit connect to the local RDB when starting an SQL session. When the CONNECT statement is completed, a message shows the relational database connection that was established. If starting a new session and COMMIT(*NONE) was not specified, or if restoring a saved session and the commit level saved with the session was not *NONE, the connection will be registered with commitment control. This implicit connect and possible commitment control registration may influence subsequent connections to remote databases. For further information, see "Determining connection type" on page 340. It is recommended that prior to connecting to the remote system:

- When connecting to an application server that does not support distributed unit of work, a RELEASE ALL followed by a COMMIT be issued to end previous connections, including the implicit connection to local.

- When connecting to a non-DB2 UDB for iSeries application server, a RELEASE ALL followed by a COMMIT be issued to end previous connections, including the implicit connection to local, and change the commitment control level to at least *CHG.

When you are connecting to a non-DB2 UDB for iSeries application server, some session attributes are changed to attributes that are supported by that application server. The following table shows the attributes that change.

Table 36. Values Table

Session Attribute	Original Value	New Value
Date Format	*YMD *DMY *MDY *JUL	*ISO *EUR *USA *USA
Time Format	*HMS with a : separator *HMS with any other separator	*JIS *EUR
Commitment Control	*CHG, *NONE *ALL	*CS Repeatable Read
Naming Convention	*SYS	*SQL
Allow Copy Data	*NO, *YES	*OPTIMIZE
Data Refresh	*ALWAYS	*FORWARD
Decimal Point	*SYSVAL	*PERIOD
Sort Sequence	Any value other than *HEX	*HEX

Notes:

1. If connecting to an server that is running a release prior to Version 2 Release 3, the sort sequence value changes to *HEX.
2. When connecting to a DB2/2 or DB2/6000 application server, the date and time formats specified must be the same format.

After the connection is completed, a message is sent stating that the session attributes have been changed. The changed session attributes can be displayed by using the session services display. While interactive SQL is running, no other connection can be established for the default activation group.

When connected to a remote system with interactive SQL, a statement processing mode of syntax-only checks the syntax of the statement against the syntax supported by the local system instead of the remote system. Similarly, the SQL prompter and list support use the statement syntax and naming conventions supported by the local system. The statement is run, however, on the remote system. Because of differences in the level of SQL support between the two systems, syntax errors may be found in the statement on the remote system at run time.

Lists of schemas and tables are available when you are connected to the local relational database. Lists of columns are available only when you are connected to a relational database manager that supports the DESCRIBE TABLE statement.

When you exit interactive SQL with connections that have pending changes or connections that use protected conversations, the connections remain. If you do not perform additional work over the connections, the connections are ended during

the next COMMIT or ROLLBACK operation. You can also end the connections by doing a RELEASE ALL and a COMMIT before exiting interactive SQL.

Using interactive SQL for remote access to non-DB2 UDB for iSeries application servers can require some setup. For more information, see the Distributed Database Programming book.

Note: In the output of a communications trace, there may be a reference to a 'CREATE TABLE XXX' statement. This is used to determine package existence; it is part of normal processing, and can be ignored.

Chapter 18. Using the SQL Statement Processor

This section describes the SQL Statement processor. This processor is available when you use the Run SQL Statements (RUNSQLSTM) command.

The SQL statement processor allows SQL statements to be executed from a source member. The statements in the source member can be run repeatedly, or changed, without compiling the source. This makes the setup of a database environment easier. The statements that can be used with the SQL statement processor are:

- ALTER TABLE
- CALL
- COMMENT ON
- COMMIT
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- GRANT (Function or Procedure Privileges)
- GRANT (Package Privileges)
- GRANT (Table Privileges)
- GRANT (User-Defined Type Privileges)
- INSERT
- LABEL ON
- LOCK TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE (Function or Procedure Privileges)
- REVOKE (Package Privileges)
- REVOKE (Table Privileges)
- REVOKE (User-Defined Type Privileges)
- ROLLBACK
- SAVEPOINT
- SET PATH
- SET SCHEMA
- SET TRANSACTION
- UPDATE

In the source member, statements end with a semicolon and do not begin with EXEC SQL. If the record length of the source member is longer than 80, only the first 80 characters will be read. Comments in the source member can be either line comments or block comments. Line comments begin with a double hyphen (--) and end at the end of the line. Block comments start with /* and can continue across many lines until the next */ is reached. Block comments can be nested. Only SQL statements and comments are allowed in the source file. The output listing and the resulting messages for the SQL statements are sent to a print file. The default print file is QSYSPRT.

To perform syntax checking only on all statements in the source member, specify the PROCESS(*SYN) parameter on the RUNSQLSTM command.

For more details, see the following sections:

- “Execution of statements after errors occur”
- “Commitment control in the SQL statement processor”
- “Schemas in the SQL Statement Processor”
- “Source member listing for the SQL statement processor” on page 301

Execution of statements after errors occur

When a statement returns an error with a severity higher than the value specified for the error level (ERRLVL) parameter of the RUNSQLSTM command, the statement has failed. The rest of the statements in the source will be parsed to check for syntax errors, but those statements will not be executed. Most SQL errors have a severity of 30. If you want to continue processing after an SQL statement fails, set the ERRLVL parameter of the RUNSQLSTM command to 30 or higher.

Commitment control in the SQL statement processor

A commitment-control level is specified on the RUNSQLSTM command. If a commitment-control level other than *NONE is specified, the SQL statements are run under commitment control. If all of the statements successfully execute, a COMMIT is done at the completion of the SQL statement processor. Otherwise, a ROLLBACK is done. A statement is considered successful if its return code severity is less than or equal to the value specified on the ERRLVL parameter of the RUNSQLSTM command.

The SET TRANSACTION statement can be used within the source member to override the level of commitment control specified on the RUNSQLSTM command.

Note: The job must be at a unit of work boundary to use the SQL statement processor with commitment control.

Schemas in the SQL Statement Processor

The SQL statement processor supports the CREATE SCHEMA statement. This is a complex statement that can be thought of as having two distinct sections. The first section defines the collection for the schema. The second section contains DDL statements that define the objects in the collection.

The first section can be written in one of two ways:

- CREATE SCHEMA *collection-name*

A collection is created using the specified collection name.

- CREATE SCHEMA AUTHORIZATION authorization-name
A collection is created using the authorization name as the collection name. When the schema is run, the user must have authority to the user profile that is named **authorization-name**.
The privileges held by the authorization-name of the statement must include:
 - Authority to run the CREATE COLLECTION statement
 - Authority to run each SQL statement within the CREATE SCHEMA

The second section of the CREATE SCHEMA statement can contain from zero to any number of the following statements:

- COMMENT ON
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE INDEX
- CREATE TABLE
- CREATE VIEW
- GRANT (Table Privileges)
- GRANT (User-Defined Type Privileges)
- LABEL ON

These statements follow directly after the first section of the statement. The statements and sections are **not** separated by semicolons. If other SQL statements follow this schema definition, the last statement in the schema must be ended by a semicolon.

All objects created or referenced in the second part of the schema statement must be in the collection that was created for the schema. All unqualified references are implicitly qualified by the collection that was created. All qualified references must be qualified by the created collection.

Source member listing for the SQL statement processor

See “Code disclaimer information” on page x information for information pertaining to code examples.

```

5722ST1 V5R2M0 020719          Run SQL Statements          SCHEMA          08/06/02 15:35:18  Page 1
Source file.....CORPDATA/SRC
Member.....SCHEMA
Commit.....*NONE
Naming.....*SYS
Generation level.....10
Date format.....*JOB
Date separator.....*JOB
Time format.....*HMS
Time separator.....*JOB
Default Collection.....*NONE
IBM SQL flagging.....*NOFLAG
ANS flagging.....*NONE
Decimal point.....*JOB
Sort Sequence.....*JOB
Language ID.....*JOB
Printer file.....*LIBL/QSYSPRT
Source file CCSID.....65535
Job CCSID.....0
Statement processing.....*RUN
Allow copy of data.....*OPTIMIZE
Allow blocking.....*READ
Source member changed on 04/01/98 11:54:10

```

Figure 8. QSYSPRT listing for SQL statement processor (Part 1 of 3)

```

5722ST1 V5R2M0 020719          Run SQL Statements          SCHEMA          08/06/02 15:35:18  Page 2
Record *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  SEQNBR Last change
1
2  DROP COLLECTION DEPT;
3  DROP COLLECTION MANAGER;
4
5  CREATE SCHEMA DEPT
6      CREATE TABLE EMP (EMPNAME CHAR(50), EMPNBR INT)
7          -- EMP will be created in collection DEPT
8      CREATE INDEX EMPIND ON EMP(EMPNBR)
9          -- EMPIND will be created in DEPT
10     GRANT SELECT ON EMP TO PUBLIC; -- grant authority
11
12     INSERT INTO DEPT/EMP VALUES('JOHN SMITH', 1234);
13         /* table must be qualified since no
14         longer in the schema */
15
16     CREATE SCHEMA AUTHORIZATION MANAGER
17         -- this schema will use MANAGER's
18         -- user profile
19     CREATE TABLE EMP_SALARY (EMPNBR INT, SALARY DECIMAL(7,2),
20         LEVEL CHAR(10))
21     CREATE VIEW LEVEL AS SELECT EMPNBR, LEVEL
22         FROM EMP_SALARY
23     CREATE INDEX SALARYIND ON EMP_SALARY(EMPNBR,SALARY)
24
25     GRANT ALL ON LEVEL TO JONES GRANT SELECT ON EMP_SALARY TO CLERK
26         -- Two statements can be on the same line
***** E N D   O F   S O U R C E *****

```

Figure 8. QSYSPRT listing for SQL statement processor (Part 2 of 3)

```

5722ST1 V5R2M0 020719          Run SQL Statements          SCHEMA          08/06/02 15:35:18  Page    3
Record *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  SEQNBR Last change
MSG ID  SEV  RECORD  TEXT
SQL7953  0      1  Position 1 Drop of DEPT in QSYS complete.
SQL7953  0      3  Position 3 Drop of MANAGER in QSYS complete.
SQL7952  0      5  Position 3 Collection DEPT created.
SQL7950  0      6  Position 8 Table EMP created in DEPT.
SQL7954  0      8  Position 8 Index EMPIND created in DEPT on table EMP in
DEPT.
SQL7966  0     10  Position 8 GRANT of authority to EMP in DEPT completed.
SQL7956  0     10  Position 40 1 rows inserted in EMP in DEPT.
SQL7952  0     13  Position 28 Collection MANAGER created.
SQL7950  0     19  Position 9 Table EMP_SALARY created in collection
MANAGER.
SQL7951  0     21  Position 9 View LEVEL created in MANAGER.
SQL7954  0     23  Position 9 Index SALARYIND created in MANAGER on table
EMP_SALARY in MANAGER.
SQL7966  0     25  Position 9 GRANT of authority to LEVEL in MANAGER
completed.
SQL7966  0     25  Position 37 GRANT of authority to EMP_SALARY in MANAGER
completed.

Message Summary
Total  Info  Warning  Error  Severe  Terminal
   13   13      0      0      0      0
00 level severity errors found in source
***** E N D   O F   L I S T I N G   * * * * *

```

Figure 8. QSYSPRT listing for SQL statement processor (Part 3 of 3)

Chapter 19. DB2 UDB for iSeries Data Protection

This chapter describes the security plan for protecting SQL data from unauthorized users and the methods for ensuring data integrity. For more information, see the following topics:

- “Security for SQL objects”
- “Securing data using iSeries Navigator” on page 307
- “Data integrity” on page 308

See “Code disclaimer information” on page x information for information pertaining to code examples.

Security for SQL objects

All objects on the server, including SQL objects, are managed by the system security function. Users may authorize SQL objects through either the SQL GRANT and REVOKE statements or the CL commands Edit Object Authority (EDTOBJAUT), Grant Object Authority (GRTOBJAUT), and Revoke Object Authority (RVKOBJAUT). For more information about system security and the use of the GRTOBJAUT and RVKOBJAUT commands, see the iSeries Security

Reference  book.

The SQL GRANT and REVOKE statements operate on SQL packages, SQL procedures, tables, views, and the individual columns of tables and views. Furthermore, SQL GRANT and REVOKE statements only grant private and public authorities. In some cases, it is necessary to use EDTOBJAUT, GRTOBJAUT, and RVKOBJAUT to authorize users to other objects, such as commands and programs.

For more information about the GRANT and REVOKE statements, see the SQL Reference book.

The authority checked for SQL statements depends on whether the statement is static, dynamic, or being run interactively.

For static SQL statements:

- If the USRPRF value is *USER, the authority to run the SQL statement locally is checked using the user profile of the user running the program. The authority to run the SQL statement remotely is checked using the user profile at the application server. *USER is the default for system (*SYS) naming.
- If the USRPRF value is *OWNER, the authority to run the SQL statement locally is checked using the user profiles of the user running the program and of the owner of the program. The authority to run the SQL statement remotely is checked using the user profiles of the application server job and the owner of the SQL package. The higher authority is the authority that is used. *OWNER is the default for SQL (*SQL) naming.

For dynamic SQL statements:

- If the USRPRF value is *USER, the authority to run the SQL statement locally is checked using the user profile of the person running the program. The authority to run the SQL statement remotely is checked using the user profile of the application server job.
- If the USRPRF value is *OWNER and DYNUSRPRF is *USER, the authority to run the SQL statement locally is checked using the user profile of the person running the program. The authority to run the SQL statement remotely is checked using the user profile of the application server job.
- If the USRPRF value is *OWNER and DYNUSRPRF is *OWNER, the authority to run the SQL statement locally is checked using the user profiles of the user running the program and the owner of the program. The authority to run the SQL statement remotely is checked using the user profiles of the application server job and the owner of the SQL package. The highest authority is the authority that is used. Because of security concerns, you should use the *OWNER parameter value for DYNUSRPRF carefully. This option gives the access authority of the owner program or package to those who run the program.

For interactive SQL statements, authority is checked against the authority of the person processing the statement. Adopted authority is not used for interactive SQL statements.

Authorization ID

The authorization ID identifies a unique user and is a user profile object on the server. Authorization IDs can be created using the system Create User Profile (CRTUSRPRF) command.

Views

A view can prevent unauthorized users from having access to sensitive data. The application program can access the data it needs in a table, without having access to sensitive or restricted data in the table. A view can restrict access to particular columns by not specifying those columns in the SELECT list (for example, employee salaries). A view can also restrict access to particular rows in a table by specifying a WHERE clause (for example, allowing access only to the rows associated with a particular department number).

Auditing

DB2 UDB for iSeries is designed to comply with the U.S. government C2 security level. A key feature of that level is the ability to audit actions on the system. DB2 UDB for iSeries uses the audit facilities managed by the system security function. Auditing can be performed on an object level, user, or system level. The system value QAUDCTL controls whether auditing is performed at the object or user level. The Change User Audit (CHGUSRAUD) command and Change Object Audit (CHGOBJAUD) command specify which users and objects are audited. The system value QAUDLVL controls what types of actions are audited (for example, authorization failures, creates, deletes, grants, revokes, etc.) For more information

about auditing see the iSeries Security Reference  book.

DB2 UDB for iSeries can also audit row changes by using the DB2 UDB for iSeries journal support.

In some cases, entries in the auditing journal will not be in the same order as they occurred. For example, a job that is running under commitment control deletes a

table, creates a new table with the same name as the one that was deleted, then does a commit. This will be recorded in the auditing journal as a create followed by a delete. This is because objects that are created are journaled immediately. An object that is deleted under commitment control is hidden and not actually deleted until a commit is done. Once the commit is done, the action is journaled.

Securing data using iSeries Navigator

You can secure your data using iSeries Navigator by:

- “Defining public authority for an object”
- “Setting up default public authority for new objects”
- “Authorizing a user or group to an object”

Defining public authority for an object

Public authority is defined for every object on the system to describe what type of access a user who does not have specific access to an object. To define public authority:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Navigate until the object for which you want to edit permissions is visible.
3. Right-click the object for which you want to add permissions and select **Permissions**.
4. On the **Permissions** dialog, select **Public** from the group list.
5. Click the **Details** button to implement detailed permissions.
6. Apply the desired permissions for the public by checking the box by the appropriate check box.
7. Click **OK**.

Setting up default public authority for new objects

Setting a default public authority allows you to have a common authority that is assigned to all new objects when they are created in library. You can edit the permissions for individual objects that require a different level of security. To set a default public authority:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Right-click on the library for which you want to set a public authority and select **Permissions**.
3. On the **Permissions** dialog, click **New Object**.
4. On the **New Object** dialog, select a default public authority. To assign an Authorization List, you can enter or **Browse** for the name of the authorization list. To view an Authorization lists properties, select **Open**.
5. Click **OK**.

Authorizing a user or group to an object

Some users may require different authority to an object than the permissions allowed by Public authority. To authorize a user or group to an object:

1. In the **iSeries Navigator** window, expand your server → **Databases** → the database that you want to work with → **Libraries**.
2. Navigate until the object for which you want to edit permissions is visible.

3. Right-click the object for which you want to add permissions and select **Permissions**.
4. On the **Permissions** dialog, click **Add**.
5. On the **Add** dialog, select one or more users and groups or enter the name of a user or group in the user or group name field.
6. Click **OK**.

This will add the users or groups to the top of the list.

Note: The user or group is given the default authority to the object. You can change a user's authority to one of the types defined by the system or you can customize the authority.

To remove user authorization from an object:


1. Select the user or group that you want to remove.
2. Click **Remove**.

Data integrity

Data integrity protects data from being destroyed or changed by unauthorized persons, system operation or hardware failures (such as physical damage to a disk), programming errors, interruptions before a job is completed (such as a power failure), or interference from running applications at the same time (such as serialization problems). Data integrity is ensured by the following functions:

- "Concurrency"
- "Journaling" on page 310
- "Commitment control" on page 311
- "Atomic operations" on page 316
- "Constraints" on page 317
- "Save/Restore" on page 318
- "Damage tolerance" on page 319
- "Index recovery" on page 319
- "Catalog integrity" on page 320
- "User auxiliary storage pool (ASP)" on page 320
- "Independent auxiliary storage pool (IASP)" on page 321

The Commitment control topic, Journal Management topic, the Database

Programming book, and the Backup and Recovery  book contain more information about each of these functions.

Concurrency

Concurrency is the ability for multiple users to access and change data in the same table or view at the same time without risk of losing data integrity. This ability is automatically supplied by the DB2 UDB for iSeries database manager. Locks are implicitly acquired on tables and rows to protect concurrent users from changing the same data at precisely the same time.

Typically, DB2 UDB for iSeries will acquire locks on rows to ensure integrity. However, some situations require DB2 UDB for iSeries to acquire a more exclusive table level lock instead of row locks. For more information, see "Commitment control" on page 311.

For example, an update (exclusive) lock on a row currently held by one cursor can be acquired by another cursor in the same program (or in a DELETE or UPDATE statement not associated with the cursor). This will prevent a positioned UPDATE or positioned DELETE statement that references the first cursor until another FETCH is performed. A read (shared no-update) lock on a row currently held by one cursor will not prevent another cursor in the same program (or DELETE or UPDATE statement) from acquiring a lock on the same row.

Default and user-specifiable lock-wait time-out values are supported. DB2 UDB for iSeries creates tables, views, and indexes with the default record wait time (60 seconds) and the default file wait time (*IMMED). This lock wait time is used for DML statements. You can change these values by using the CL commands Change Physical File (CHGPF), Change Logical File (CHGLF), and Override Database File (OVRDBF).

The lock wait time used for all DDL statements and the LOCK TABLE statement, is the job default wait time (DFTWAIT). You can change this value by using the CL commands Change Job (CHGJOB) or Change Class (CHGCLS).

In the event that a large record wait time is specified, deadlock detection is provided. For example, assume one job has an exclusive lock on row 1 and another job has an exclusive lock on row 2. If the first job attempts to lock row 2, it will wait because the second job is holding the lock. If the second job then attempts to lock row 1, DB2 UDB for iSeries will detect that the two jobs are in a deadlock and an error will be returned to the second job.

You can explicitly prevent other users from using a table at the same time by using the SQL LOCK TABLE statement, described in the SQL Reference book. Using COMMIT(*RR) will also prevent other users from using a table during a unit of work.

In order to improve performance, DB2 UDB for iSeries will frequently leave the open data path (ODP) open (for details, see the Database Performance and Query Optimization information). This performance feature also leaves a lock on tables referenced by the ODP, but does not leave any locks on rows. A lock left on a table may prevent another job from performing an operation on that table. In most cases, however, DB2 UDB for iSeries will detect that other jobs are holding locks and events will be signalled to those jobs. The event causes DB2 UDB for iSeries to close any ODPs (and release the table locks) that are associated with that table and are currently only open for performance reasons. Note that the lock wait time out must be large enough for the events to be signalled and the other jobs to close the ODPs or an error will be returned.

Unless the LOCK TABLE statement is used to acquire table locks, or either COMMIT(*ALL) or COMMIT(*RR) is used, data which has been read by one job can be immediately changed by another job. Usually, the data that is read at the time the SQL statement is executed and therefore it is very current (for example, during FETCH). In the following cases, however, data is read prior to the execution of the SQL statement and therefore the data may not be current (for example, during OPEN).

- ALWCPYDTA(*OPTIMIZE) was specified and the optimizer determined that making a copy of the data would perform better than not making a copy.
- Some queries require the database manager to create a temporary result table. The data in the temporary result table will not reflect changes made after the cursor was opened. A temporary result table is required when:

- The total length in bytes of storage for the columns specified in an ORDER BY clause exceeds 2000 bytes.
 - ORDER BY and GROUP BY clauses specify different columns or columns in a different order.
 - UNION or DISTINCT clauses are specified.
 - ORDER BY or GROUP BY clauses specify columns which are not all from the same table.
 - Joining a logical file defined by the JOINDFT data definition specifications (DDS) keyword with another file.
 - Joining or specifying GROUP BY on a logical file which is based on multiple database file members.
 - The query contains a join in which at least one of the files is a view which contains a GROUP BY clause.
 - The query contains a GROUP BY clause which references a view that contains a GROUP BY clause.
- A basic subquery is evaluated when the query is opened.

Journaling

The DB2 UDB for iSeries journal support supplies an audit trail and forward and backward recovery. Forward recovery can be used to take an older version of a table and apply the changes logged on the journal to the table. Backward recovery can be used to remove changes logged on the journal from the table.

When an SQL schema is created, a journal and journal receiver are created in the schema. When SQL creates the journal and journal receiver, they are only created on a user auxiliary storage pool (ASP) if the ASP clause is specified on the CREATE COLLECTION or the CREATE SCHEMA statement. However, because placing journal receivers on their own ASPs can improve performance, the person managing the journal might want to create all future journal receivers on a separate ASP.

When a table is created into the schema, it is automatically journaled to the journal DB2 UDB for iSeries created in the schema (QSQRN). A table created in a non-schema will also have journaling started if a journal named QSQRN exists in that library. After this point, it is your responsibility to use the journal functions to manage the journal, the journal receivers, and the journaling of tables to the journal. For example, if a table is moved into a schema, no automatic change to the journaling status occurs. If a table is restored, the normal journal rules apply. That is, if the table was journaled at the time of the save, it is journaled to the same journal at restore time. If the table was not journaled at the time of the save, it is not journaled at restore time.

The journal created in the SQL collection is normally the journal used for logging all changes to SQL tables. You can, however, use the system journal functions to journal SQL tables to a different journal.

A user can stop journaling on any table using the journal functions, but doing so prevents an application from running under commitment control. If journaling is stopped on a parent table of a referential constraint with a delete rule of NO ACTION, CASCADE, SET NULL, or SET DEFAULT, all update and delete operations will be prevented. Otherwise, an application is still able to function if you have specified COMMIT(*NONE); however, this does not provide the same level of integrity that journaling and commitment control provide.

For more information about journaling, see the Journaling topic.

Commitment control

The DB2 UDB for iSeries commitment control support provides a means to process a group of database changes (updates, inserts, DDL operations, or deletes) as a single unit of work (transaction). A commit operation guarantees that the group of operations is completed. A rollback operation guarantees that the group of operations is backed out. A savepoint can be used to break a transaction into smaller units that can be rolled back. A commit operation can be issued through several different interfaces. For example,

- An SQL COMMIT statement
- A CL COMMIT command
- A language commit statement (such as an RPG COMMIT statement)

A rollback operation can be issued through several different interfaces. For example,

- An SQL ROLLBACK statement
- A CL ROLLBACK command
- A language rollback statement (such as an RPG ROLBK statement)

The only SQL statements that cannot be committed or rolled back are:

- DROP COLLECTION
- GRANT or REVOKE if an authority holder exists for the specified object

If commitment control was not already started when either an SQL statement is executed with an isolation level other than COMMIT(*NONE) or a RELEASE statement is executed, then DB2 UDB for iSeries sets up the commitment control environment by implicitly calling the CL command Start Commitment Control (STRCMTCTL). DB2 UDB for iSeries specifies NFYOBJ(*NONE) and CMTSCOPE(*ACTGRP) parameters along with LCKLVL on the STRCMTCTL command. The LCKLVL specified is the lock level on the COMMIT parameter on the CRTSQLxxx, STRSQL, or RUNSQLSTM commands. In REXX, the LCKLVL specified is the lock level on the SET OPTION statement. You may use the STRCMTCTL command to specify a different CMTSCOPE, NFYOBJ, or LCKLVL. If you specify CMTSCOPE(*JOB) to start the job level commitment definition, DB2 UDB for iSeries uses the job level commitment definition for programs in that activation group.

Notes:

1. When using commitment control, the tables referred to in the application program by Data Manipulation Language statements must be journaled.
2. Note that the LCKLVL specified is only the default lock level. After commitment control is started, the SET TRANSACTION SQL statement and the lock level specified on the COMMIT parameter on the CRTSQLxxx, STRSQL, or RUNSQLSTM commands will override the default lock level.

For cursors that use column functions, GROUP BY, or HAVING, and are running under commitment control, a ROLLBACK HOLD has no effect on the cursor's position. In addition, the following occurs under commitment control:

- If COMMIT(*CHG) and (ALWBLK(*NO) or (ALWBLK(*READ))) is specified for one of these cursors, a message (CPI430B) is sent that says COMMIT(*CHG) requested but not allowed.

- If COMMIT(*ALL), COMMIT(*RR), or COMMIT(*CS) with the KEEP LOCKS clause is specified for one of the cursors, DB2 UDB for iSeries will lock all referenced tables in shared mode (*SHRNUP). The lock prevents concurrent application processes from executing any but read-only operations on the named table. A message (either SQL7902 or CPI430A) is sent that says COMMIT(*ALL), COMMIT(*RR), or COMMIT(*CS) with the KEEP LOCKS clause is specified for one of the cursors requested but not allowed. Message SQL0595 may also be sent.

For cursors where either COMMIT(*ALL), COMMIT(*RR), or COMMIT(*CS) with the KEEP LOCKS clause is specified and either catalog files are used or a temporary result table is required, DB2 UDB for iSeries will lock all referenced tables in shared mode (*SHRNUP). This will prevent concurrent processes from executing anything but read-only operations on the table(s). A message (either SQL7902 or CPI430A) is sent that says COMMIT(*ALL) is requested but not allowed. Message SQL0595 may also be sent.

If ALWBLK(*ALLREAD) and COMMIT(*CHG) were specified, when the program was precompiled, all read only cursors will allow blocking of rows and a ROLLBACK HOLD will not roll the cursor position back.

If COMMIT(*RR) is requested, the tables will be locked until the query is closed. If the cursor is read only, the table will be locked (*SHRNUP). If the cursor is in update mode, the table will be locked (*EXCLRD). Since other users will be locked out of the table, running with repeatable read will prevent concurrent access of the table.

If an isolation level other than COMMIT(*NONE) was specified and the application issues a ROLLBACK or the activation group ends abnormally (and the commitment definition is not *JOB), all updates, inserts, deletes, and DDL operations made within the unit of work are backed out. If the application issues a COMMIT or the activation group ends normally, all updates, inserts, deletes, and DDL operations made within the unit of work are committed.

DB2 UDB for iSeries uses locks on rows to keep other jobs from accessing changed data before a unit of work completes. If COMMIT(*ALL) is specified, read locks on rows fetched are also used to prevent other jobs from changing data that was read before a unit of work completes. This will not prevent other jobs from reading the unchanged rows. This ensures that, if the same unit of work rereads a row, it gets the same result. Read locks do not prevent other jobs from fetching the same rows.

Commitment control handles up to 500 million distinct row changes in a unit of work. If COMMIT(*ALL) or COMMIT(*RR) is specified, all rows read are also included in the limit. (If a row is changed or read more than once in a unit of work, it is only counted once toward the limit.) Holding a large number of locks adversely affects system performance and does not allow concurrent users to access rows locked in the unit of work until the end of the unit of work. It is in your best interest to keep the number of rows processed in a unit of work small.

Commitment control will allow up to 512 files for each journal to be open under commitment control or closed with pending changes in a unit of work.

COMMIT HOLD and ROLLBACK HOLD allows you to keep the cursor open and start another unit of work without issuing an OPEN again. The HOLD value is not available when you are connected to a remote database that is not on an iSeries system. However, the WITH HOLD option on DECLARE CURSOR may be used to

keep the cursor open after a COMMIT. This type of cursor is supported when you are connected to a remote database that is not on an iSeries system. Such a cursor is closed on a rollback.

Table 37. Row Lock Duration

SQL Statement	COMMIT Parameter (See note 5)	Duration of Row Locks	Lock Type
SELECT INTO SET variable VALUES INTO	*NONE *CHG *CS (See note 7) *ALL (See note 2)	No locks No locks Row locked when read and released From read until ROLLBACK or COMMIT	READ READ
FETCH (read-only cursor)	*NONE *CHG *CS (See note 7) *ALL (See note 2)	No locks No locks From read until the next FETCH From read until ROLLBACK or COMMIT	READ READ
FETCH (update or delete capable cursor) (See note 1)	*NONE *CHG *CS *ALL	When row not updated or deleted from read until next FETCH When row is updated or deleted from read until UPDATE or DELETE When row not updated or deleted from read until next FETCH When row is updated or deleted from read until COMMIT or ROLLBACK When row not updated or deleted from read until next FETCH When row is updated or deleted from read until COMMIT or ROLLBACK From read until ROLLBACK or COMMIT	UPDATE UPDATE UPDATE UPDATE
INSERT (target table)	*NONE *CHG *CS *ALL	No locks From insert until ROLLBACK or COMMIT From insert until ROLLBACK or COMMIT From insert until ROLLBACK or COMMIT	UPDATE UPDATE UPDATE ³
INSERT (tables in subselect)	*NONE *CHG *CS *ALL	No locks No locks Each row locked while being read From read until ROLLBACK or COMMIT	READ READ
UPDATE (non-cursor)	*NONE *CHG *CS *ALL	Each row locked while being updated From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT	UPDATE UPDATE UPDATE UPDATE
DELETE (non-cursor)	*NONE *CHG *CS *ALL	Each row locked while being deleted From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT	UPDATE UPDATE UPDATE UPDATE
UPDATE (with cursor)	*NONE *CHG *CS *ALL	Lock released when row updated From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT	UPDATE UPDATE UPDATE UPDATE
DELETE (with cursor)	*NONE *CHG *CS *ALL	Lock released when row deleted From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT From read until ROLLBACK or COMMIT	UPDATE UPDATE UPDATE UPDATE
Subqueries (update or delete capable cursor or UPDATE or DELETE non-cursor)	*NONE *CHG *CS *ALL (see note 2)	From read until next FETCH From read until next FETCH From read until next FETCH From read until ROLLBACK or COMMIT	READ READ READ READ

Table 37. Row Lock Duration (continued)

SQL Statement	COMMIT Parameter (See note 5)	Duration of Row Locks	Lock Type
Subqueries (read-only cursor or SELECT INTO)	*NONE *CHG *CS *ALL	No locks No locks Each row locked while being read From read until ROLLBACK or COMMIT	READ READ
<p>Notes:</p> <ol style="list-style-type: none"> 1. A cursor is open with UPDATE or DELETE capabilities if the result table is not read-only (see description of DECLARE CURSOR in SQL Reference book) and if one of the following is true: <ul style="list-style-type: none"> • The cursor is defined with a FOR UPDATE clause. • The cursor is defined without a FOR UPDATE, FOR READ ONLY, or ORDER BY clause and the program contains at least one of the following: <ul style="list-style-type: none"> – Cursor UPDATE referring to the same cursor-name – Cursor DELETE referring to the same cursor-name – An EXECUTE or EXECUTE IMMEDIATE statement and ALWBLK(*READ) or ALWBLK(*NONE) was specified on the CRTSQLxxx command. 2. A table or view can be locked exclusively in order to satisfy COMMIT(*ALL). If a subselect is processed that includes a UNION, or if the processing of the query requires the use of a temporary result, an exclusive lock is acquired to protect you from seeing uncommitted changes. 3. An UPDATE lock on rows of the target table and a READ lock on the rows of the subselect table. 4. A table or view can be locked exclusively in order to satisfy repeatable read. Row locking is still done under repeatable read. The locks acquired and their duration are identical to *ALL. 5. Repeatable read (*RR) row locks will be the same as the locks indicated for *ALL. 6. For a detailed explanation of isolation levels and locking, see Isolation Level in the SQL Reference book. 7. If the KEEP LOCKS clause is specified with *CS, any read locks are held until the cursor is closed or until a COMMIT or ROLLBACK is done. If no cursors are associated with the isolation clause, then locks are held until the completion of the SQL statement. 			

For more details about commitment control, see the Commitment control topic.

Savepoints

Savepoints allow you to create milestones within a transaction. If the transaction rolls back, changes are undone back to the specified savepoint, rather than to the beginning of the transaction. A savepoint is set by using the SAVEPOINT SQL statement. For example, create a savepoint called STOP_HERE:

```
SAVEPOINT STOP_HERE
ON ROLLBACK RETAIN CURSORS
```

Program logic in the application dictates whether the savepoint name is reused as the application progresses, or if the savepoint name denotes a unique milestone in the application that should not be reused.

If the savepoint represents a unique milestone that should not be moved with another SAVEPOINT statement, specify the UNIQUE keyword. This prevents the accidental reuse of the name that could occur by invoking a stored procedure that uses the identical savepoint name in a SAVEPOINT statement. However, if the SAVEPOINT statement is used in a loop, then the UNIQUE keyword should not be used. The following SQL statement sets a unique savepoint named START_OVER.

```
SAVEPOINT START_OVER UNIQUE
ON ROLLBACK RETAIN CURSORS
```


To rollback to a savepoint, use the ROLLBACK statement with the TO SAVEPOINT clause. The following example illustrates using the SAVEPOINT and ROLLBACK TO SAVEPOINT statements:

This application logic books airline reservations on a preferred date, then books hotel reservations. If the hotel is unavailable, it rolls back the airline reservations and then repeats the process for another date. Up to 3 dates are tried.

```
got_reservations =0;
EXEC SQL SAVEPOINT START_OVER UNIQUE ON ROLLBACK RETAIN CURSORS;

if (SQLCODE != 0) return;

for (i=0; i<3 & got_reservations == 0; ++i)
{
  Book_Air(dates(i), ok);
  if (ok)
  {
    Book_Hotel(dates(i), ok);
    if (ok) got_reservations = 1;
    else
    {
      EXEC SQL ROLLBACK TO SAVEPOINT START_OVER;
      if (SQLCODE != 0) return;
    }
  }
}

EXEC SQL RELEASE SAVEPOINT START_OVER;
```

Savepoints are released using the RELEASE SAVEPOINT statement. If a RELEASE SAVEPOINT statement is not used to explicitly release a savepoint, it is released at the end of the current savepoint level or at the end of the transaction. The following statement releases savepoint START_OVER.

```
RELEASE SAVEPOINT START_OVER
```

Savepoints are released when the transaction is committed or rolled back. Once the savepoint name is released, a rollback to the savepoint name is no longer possible. The COMMIT or ROLLBACK statement releases all savepoint names established within a transactions. Since all savepoint names are released within the transaction, all savepoint names can be reused following a commit or rollback.

Levels of savepoints

A single statement can implicitly or explicitly invoke a user-defined function, trigger, or stored procedure. This is known as nesting. In some cases when a new nesting level is initiated, a new savepoint level is also initiated. A new savepoint level isolates the invoking application from any savepoint activity by the lower level routine or trigger.

Savepoints can only be referenced within the same savepoint level (or scope) in which they are defined. A ROLLBACK TO SAVEPOINT statement cannot be used to rollback to a savepoint established outside the current savepoint level. Likewise, a RELEASE SAVEPOINT statement cannot be used to release a savepoint established outside the current savepoint level. The following table summarizes when savepoint levels are initiated and terminated:

A new savepoint level is initiated when:	That savepoint level terminates when:
A new unit of work is started	COMMIT or ROLLBACK is issued
A trigger is invoked	The trigger completes

A new savepoint level is initiated when:	That savepoint level terminates when:
A user-defined function is invoked	The user-defined function returns to the invoker
A stored procedure is invoked, and that stored procedure was created with the NEW SAVEPOINT LEVEL clause	The stored procedure returns to the caller
There is a BEGIN for an ATOMIC compound SQL statement	There is an END for an ATOMIC compound statement

A savepoint that is established in a savepoint level is implicitly released when that savepoint level is terminated.

Considerations for Savepoints within a Distributed Database Environment

Savepoints are scoped to a single connection only. Once a savepoint is established, it is not distributed to all remote databases that the application connects to. The savepoint only applies to the current database that the application is connected to when the savepoint is established.

Atomic operations

When running under COMMIT(*CHG), COMMIT(*CS), or COMMIT(*ALL), all operations are guaranteed to be atomic. That is, they will complete or they will appear not to have started. This is true regardless of when or how the function was ended or interrupted (such as power failure, abnormal job end, or job cancel).

If COMMIT (*NONE) is specified, however, some underlying database data definition functions are not atomic. The following SQL data definition statements are guaranteed to be atomic:

- ALTER TABLE (See note 1)
- COMMENT ON (See note 2)
- LABEL ON (See note 2)
- GRANT (See note 3)
- REVOKE (See note 3)
- DROP TABLE (See note 4)
- DROP VIEW (See note 4)
- DROP INDEX
- DROP PACKAGE

Notes:

1. If constraints need to be added or removed, as well as column definitions changed, the operations are processed one at a time, so the entire SQL statement is not atomic. The order of operation is:
 - Remove constraints
 - Drop columns for which the RESTRICT option was specified
 - All other column definition changes (DROP COLUMN CASCADE, ALTER COLUMN, ADD COLUMN)
 - Add constraints
2. If multiple columns are specified for a COMMENT ON or LABEL ON statement, the columns are processed one at a time, so the entire SQL statement is not atomic, but the COMMENT ON or LABEL ON to each individual column or object will be atomic.

3. If multiple tables, SQL packages, or users are specified for a GRANT or REVOKE statement, the tables are processed one at a time, so the entire SQL statement is not atomic, but the GRANT or REVOKE to each individual table will be atomic.
4. If dependent views need to be dropped during DROP TABLE or DROP VIEW, each dependent view is processed one at a time, so the entire SQL statement is not atomic.

The following data definition statements are not atomic because they involve more than one DB2 UDB for iSeries database operation:

```

CREATE ALIAS
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE INDEX
CREATE PROCEDURE
CREATE SCHEMA
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DROP ALIAS
DROP DISTINCT TYPE
DROP FUNCTION
DROP PROCEDURE
DROP SCHEMA
DROP TRIGGER
RENAME (See note 1)

```

Notes:

1. RENAME is atomic only if the name or the system name is changed. When both are changed, the RENAME is not atomic.

For example, a CREATE TABLE can be interrupted after the DB2 UDB for iSeries physical file has been created, but before the member has been added. Therefore, in the case of create statements, if an operation ends abnormally, you may have to drop the object and then create it again. In the case of a DROP COLLECTION statement, you may have to drop the collection again or use the CL command Delete Library (DLTLIB) to remove the remaining parts of the collection.

Constraints

DB2 UDB for iSeries supports unique, referential, and check constraints. A unique constraint is a rule that guarantees that the values of a key are unique. A referential constraint is a rule that all non-null values of foreign keys in a dependent table have a corresponding parent key in a parent table. A check constraint is a rule that limits the values allowed in a column or group of columns.

DB2 UDB for iSeries will enforce the validity of the constraint during any DML (data manipulation language) statement. Certain operations (such as restore of the dependent table), however, cause the validity of the constraint to be unknown. In this case, DML statements may be prevented until DB2 UDB for iSeries has verified the validity of the constraint.

- Unique constraints are implemented with indexes. If an index that implements a unique constraint is invalid, the Edit Rebuild of Access Paths (EDTRBDAP) command can be used to display any indexes that currently require rebuild.
- If DB2 UDB for iSeries does not currently know whether a referential constraint or check constraint is valid, the constraint is considered to be in a check pending state. The Edit Check Pending Constraints (EDTCPCST) command can be used to display any indexes that currently require rebuild.

For more information about constraints, see Chapter 10, “Data Integrity” on page 135 and the Database Programming book.

Save/Restore

The OS/400 save/restore functions are used to save tables, views, indexes, journals, journal receivers, SQL packages, SQL procedures, SQL triggers, user-defined functions, user-defined types, and collections on disk (save file) or to some external media (tape or diskette). The saved versions can be restored onto any iSeries system at some later time. The save/restore function allows an entire collection, selected objects, or only objects changed since a given date and time to be saved. All information needed to restore an object to its previous state is saved. This function can be used to recover from damage to individual tables by restoring the data with a previous version of the table or the entire collection.

When a program that was created for an SQL procedure or a service program that was created for an SQL function or a sourced function is restored, it is automatically added to the SYSROUTINES and SYSPARMS catalogs, as long as the procedure or function does not already exist with the same signature. SQL programs created in QSYS will not be created as SQL procedures when restored. Additionally, external programs or service programs that were referenced on a CREATE PROCEDURE or CREATE FUNCTION statement may contain the information required to register the routine in SYSROUTINES. If the information exists and the signature is unique, the functions or procedures will also be added to SYSROUTINES and SYSPARMS when restored.

When an SQL table is restored, the definitions for the SQL triggers that are defined for the table are also restored. The SQL trigger definitions are automatically added to the SYSTRIGGERS, SYSTRIGDEP, SYSTRIGCOL, and SYSTRIGUPD catalogs. The program object that is created from the SQL CREATE TRIGGER statement must also be saved and restored when the SQL table is saved and restored. The saving and restoring of the program object is not automated by the database manager. The precautions for self-referencing triggers should be reviewed when restoring SQL tables to a new library. See Inoperative triggers in the Notes of the CREATE TRIGGER statement section of the SQL Reference book.

When an *SQLUDT object is restored for a user-defined type, the user-defined type is automatically added to the SYSTYPES catalog. The appropriate functions needed to cast between the user-defined type and the source type are also created, as long as the type and functions do not already exist.

Either a distributed SQL program or its associated SQL package can be saved and restored to any number of systems. This allows any number of copies of the SQL programs on different systems to access the same SQL package on the same application server. This also allows a single distributed SQL program to connect to any number of application servers that have the SQL package restored (CRTSQLPKG can also be used). SQL packages cannot be restored to a different library.

Attention: Restoring a schema to an existing library or to a schema that has a different name does not restore the journal, journal receivers, or IDDU dictionary (if one exists). If the schema is restored to a schema with a different name, the catalog views in that schema will only reflect objects in the old schema. The catalog views in QSYS2, however, will appropriately reflect all objects.

Damage tolerance

The server provides several mechanisms to reduce or eliminate damage caused by disk errors. For example, mirroring, checksums, and RAID disks can all reduce the possibility of disk problems. The DB2 UDB for iSeries functions also have a certain amount of tolerance to damage caused by disk errors or system errors.

A DROP operation always succeeds, regardless of the damage. This ensures that should damage occur, at least the table, view, SQL package, index, procedure, function, or distinct type can be deleted and restored or created again.

In the event that a disk error has damaged a small portion of the rows in a table, the DB2 UDB for iSeries database manager allows you to read rows still accessible.

Index recovery

DB2 UDB for iSeries supplies several functions to deal with index recovery.

- System managed index protection

The EDTRCYAP CL command allows a user to instruct DB2 UDB for iSeries to guarantee that in the event of a system or power failure, the amount of time required to recover all indexes on the system is kept below a specified time. The system automatically journals enough information in a system journal to limit the recovery time to the specified amount.

- Journaling of indexes

DB2 UDB for iSeries supplies an index journaling function that makes it unnecessary to rebuild an entire index due to a power or system failure. If the index is journaled, the system database support automatically makes sure the index is in synchronization with the data in the tables without having to rebuild it from scratch. SQL indexes are *not* journaled automatically. You can, however, use the CL command Start Journal Access Path (STRJRNAP) to journal any index created by DB2 UDB for iSeries.

- Index rebuild

All indexes on the system have a maintenance option that specifies when an index is maintained. SQL indexes are created with an attribute of *IMMED maintenance.

In the event of a power failure or abnormal system failure, if indexes were not protected by one of the previously described techniques, those indexes in the process of change may need to be rebuilt by the database manager to make sure they agree with the actual data. All indexes on the system have a recovery option that specifies when an index should be rebuilt if necessary. All SQL indexes with an attribute of UNIQUE are created with a recovery attribute of *IPL (this means that these indexes are rebuilt before the OS/400 has been started). All other SQL indexes are created with the *AFTIPL recovery option (this means that after the operating system has been started, indexes are asynchronously rebuilt). During an IPL, the operator can see a display showing indexes needing to be rebuilt and their recovery option. The operator can override the recovery options.

- Save and restore of indexes

The save/restore function allows you to save indexes when a table is saved by using ACCPTH(*YES) on the Save Object (SAVOBJ) or Save Library (SAVLIB) CL commands. In the event of a restore when the indexes have also been saved, there is no need to rebuild the indexes. Any indexes not previously saved and restored are automatically and asynchronously rebuilt by the database manager.

Catalog integrity

Catalogs contain information about tables, views, SQL packages, indexes, procedures, functions, triggers, and parameters in a schema. The database manager ensures that the information in the catalog is accurate at all times. This is accomplished by preventing end users from explicitly changing any information in the catalog and by implicitly maintaining the information in the catalog when changes occur to the tables, views, SQL packages, indexes, types, procedures, functions, triggers, and parameters described in the catalog.


The integrity of the catalog is maintained whether objects in the schema are changed by SQL statements, OS/400 CL commands, System/38 Environment CL commands, System/36 Environment functions, or any other product or utility on an iSeries system. For example, deleting a table can be done by running an SQL DROP statement, issuing an OS/400 DLTF CL command, issuing a System/38 DLTF CL command or entering option 4 on a WRKF or WRKOBJ display. Regardless of the interface used to delete the table, the database manager will remove the description of the table from the catalog at the time the delete is performed. The following is a list of functions and the associated effect on the catalog:

Table 38. Effect of Various Functions on Catalogs

Function	Effect on the Catalog
Add constraint to table	Information added to catalog
Remove of constraint from table	Related information removed from catalog
Create object into schema	Information added to catalog
Delete of object from schema	Related information removed from catalog
Restore of object into schema	Information added to catalog
Change of object long comment	Comment updated in catalog
Change of object label (text)	Label updated in catalog
Change of object owner	Owner updated in catalog
Move of object from a schema	Related information removed from catalog
Move of object into schema	Information added to catalog
Rename of object	Name of object updated in catalog

User auxiliary storage pool (ASP)

A schema can be created in a user ASP by using the ASP clause on the CREATE COLLECTION and CREATE SCHEMA statements. The CRTLIB command can also be used to create a library in a user ASP. That library can then be used to receive

SQL tables, views, and indexes. See the Backup and Recovery  book for more information about auxiliary storage pools.

Independent auxiliary storage pool (IASP)

Independent disk pools are used to set up user databases on the iSeries server. There are three types of independent disk pools: primary, secondary, and user-defined file system (UDFS). Databases are set up using primary independent disk pools.

With iSeries servers, you can work with multiple databases. The iSeries server provides a system database (often referred to as SYSBAS) and the ability to work with one or more user databases. User databases are implemented on the iSeries server through the use of independent disk pools, which are set up in the Disk Management function of iSeries Navigator. Once an independent disk pool is set up, it appears as another database under the Databases function of iSeries Navigator.

Chapter 20. Testing SQL Statements in Application Programs

This chapter describes how to establish a test environment for SQL statements in an application program and how to debug this program.

For more details, see the following sections:

- “Establishing a test environment”
- “Testing your SQL application programs” on page 324

Establishing a test environment

Some things you need to test your program are:

- **Authorization.** You need to be authorized to create tables and views, access SQL data, and create and run programs.
- **A test data structure.** If your program updates, inserts, or deletes data from tables and views, *you should use test data* to verify the running of the program. If your program only retrieves data from tables and views, you might consider using production-level data when testing your program. It is recommended, however, that you use the CL command Start Debug (STRDBG) with UPDPROD(*NO) to assure that the production level data does not accidentally get changed. For more information about debugging, see the chapter on testing

in the CL Programming  book.

- **Test input data.** The input data your program uses during testing should be valid data that represents as many possible input conditions as you can think of. You cannot be sure that your output data is valid unless you use valid input data.

If your program verifies that input data is valid, include both valid and not valid data to verify that the valid data is processed and the not valid data is detected.

You might have to refresh the data for subsequent tests.

To test the program thoroughly, test as many of the paths through the program as possible. For example:

- Use input data that forces the program to run each of its branches.
- Check the results. For example, if the program updates a row, select the row to see if it was updated correctly.
- Be sure to test the program error routines. Again, use input data that forces the program to encounter as many of the anticipated error conditions as possible.
- Test the editing and validation routines your program uses. Give the program as many different combinations of input data as possible to verify that it correctly edits or validates that data.

For more details, see “Designing a test data structure”.

Designing a test data structure

To test an application that accesses SQL data, you might have to create test tables and views:

- **Test views of existing tables.** If your application does not change data and the data exists in one or more production-level tables, you might consider using a view of the existing tables. It is also recommended that you use STRDBG command with UPDPDPROD(*NO) to assure that the production level data does not accidentally get changed. See the chapter on testing in the CL Programming



for more information about debugging.

- **Test tables.** When your application creates, changes, or deletes data, you will probably want to test the application by using tables that contain test data. See Chapter 2, “Getting Started with SQL” for a description of how to create tables and views.

Also, you might want to use the CL command Create Duplicate Object (CRTDUPOBJ) to create a duplicate test table, view, or index.

Authorization

Before you can create a table, you must be authorized to create tables and to use the schema in which the table is to reside. In addition, you must have authority to create and run the programs you want to test.

If you intend to use existing tables and views (either directly or as the basis for a view), you must be authorized to access those tables and views.

If you want to create a view, you must be authorized to create views and must have authorization to each table and view on which the view is based. For more information about specific authorities required for any specific SQL statement, see the SQL Reference book.

Testing your SQL application programs

There are two phases for testing DB2 UDB for iSeries SQL applications: the program debug phase and the performance verification phase. They are: “Program debug phase” and “Performance verification phase” on page 325.

Program debug phase

This test phase is done to ensure that the SQL queries are specified correctly and that the program is producing the correct results.

Debugging your program with SQL statements is much the same as debugging your program without SQL statements. However, when SQL statements are run in a job in the debug mode, the database manager puts messages in the job log about how each SQL statement ran. This message is an indication of the SQLCODE for the SQL statement. If the statement ran successfully, the SQLCODE value is zero, and a completion message is issued. A negative SQLCODE results in a diagnostic message. A positive SQLCODE results in an informational message.

The message is either a 4-digit code prefixed by **SQL** or a 5-digit code prefixed by **SQ**. For example, an SQLCODE of -204 results in a message of SQL0204, and an SQLCODE of 30000 results in a message of SQ30000.

Associated with a SQLCODE is a SQLSTATE. The SQLSTATE is an additional return code provided in the SQLCA that identifies common error conditions among the different IBM relational database products. The same error condition on different relational database products will produce the same SQLSTATE. The same error conditions will not produce the same SQLCODE. This return code is

particularly useful when determining the cause of errors returned from the relational database operations performed on non-DB2 UDB for iSeries system.

For non-ILE program debugging, references to high-level language statement numbers in debug mode must be taken from the compile listing. For ILE program debugging, precompile the program specifying `DBGVIEW(*SOURCE)` and then use the source-level debugger.

SQL will always put messages in the job log for negative `SQLCODEs` and positive codes other than +100 regardless of whether it is in debug mode or not.

Performance verification phase

This test phase verifies that the appropriate indexes are available and that the queries are coded in a manner that allows the database manager to resolve the queries in the expected response time. The performance of an SQL application is dependent on the attributes of the tables being accessed. If you use small tables, the response time of the query is not affected by the availability of indexes. However, when you run that same query on a database with large tables and appropriate indexes do not exist, the response time for the queries can be very long.

The test environment should resemble the production environment as closely as possible. The test schema should have tables with the same names and composition as the production schema. The same indexes need to be available over the tables in both schemas. The number of rows and the distribution of values in the tables should be similar.

See the Database Performance and Query Optimization book for a description of the tools and commands you can use to verify performance.

Chapter 21. Distributed Relational Database Function

A *distributed relational database* consists of a set of SQL objects that are spread across interconnected computer systems. These relational databases can be of the same type (for example, DB2 UDB for iSeries) or of different types (DB2 Universal Database for OS/390, DB2 for VSE and VM, DB2 Universal Database (UDB), or non-IBM database management systems which support DRDA). Each relational database has a relational database manager to manage the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a given database manager access to run SQL statements on a relational database on another system.

The application requester supports the application side of a connection. The application server is the local or remote database to which an application requester is connected. DB2 UDB for iSeries provides support for Distributed Relational Database Architecture (DRDA) to allow an application requester to communicate with application servers. In addition, DB2 UDB for iSeries can invoke exit programs to allow access to data on other database management systems which do not support DRDA. These exit programs are called application requester driver (ARD) programs.

DB2 UDB for iSeries supports two levels of distributed relational database:

- Remote unit of work (RUW)

Remote unit of work is where the preparation and running of SQL statements occurs at only one application server during a unit of work. DB2 UDB for iSeries supports RUW over either APPC or TCP/IP.

- Distributed unit of work (DUW)

Distributed unit of work is where the preparation and running of SQL statements can occur at multiple applications servers during a unit of work. However, a single SQL statement can only refer to objects located at a single application server. DB2 UDB for iSeries supports DUW over APPC and, beginning in V5R1, introduced support for DUW over TCP/IP.

For comprehensive information about distributed relational databases, see the Distributed Database Programming book.

For more details, see

- “DB2 UDB for iSeries distributed relational database support” on page 328
- “DB2 UDB for iSeries distributed relational database example program” on page 328
- “SQL package support” on page 329
- “CCSID considerations for SQL” on page 333
- “Connection management and activation groups” on page 334
- “Distributed support” on page 339
- “Distributed unit of work” on page 346
- “Application requester driver programs” on page 350
- “Problem handling” on page 351
- “DRDA stored procedure considerations” on page 351

DB2 UDB for iSeries distributed relational database support

The DB2 UDB Query Manager and SQL Development Kit licensed program supports interactive access to distributed databases with the following SQL statements:

- CONNECT
- SET CONNECTION
- DISCONNECT
- RELEASE
- DROP PACKAGE
- GRANT PACKAGE
- REVOKE PACKAGE

For detailed descriptions of these statements, see the SQL Reference book.

Additional support is provided by the development kit through parameters on the SQL precompiler commands:

- Create SQL ILE C Object (CRTSQLCI) command
- Create SQL ILE C++ Object (CRTSQLCPPI) command
- Create SQL COBOL Program (CRTSQLCBL) command
- Create SQL ILE COBOL Object (CRTSQLCBLI) command
- Create SQL PL/I Program (CRTSQLPLI) command
- Create SQL RPG Program (CRTSQLRPG) command
- Create SQL ILE RPG Object (CRTSQLRPGI) command

For more information about the SQL precompiler commands, see the topic *Preparing and Running a Program with SQL Statements in the SQL Programming with Host Languages* information. The create SQL Package (CRTSQLPKG) command allows you to create an SQL package from an SQL program that was created as a distributed program. Syntax and parameter definitions for the CRTSQLPKG and CRTSQLxxx commands are provided in Appendix B, "DB2 UDB for iSeries CL Command Descriptions".

See "Code disclaimer information" on page x information for information pertaining to code examples.

DB2 UDB for iSeries distributed relational database example program

A remote unit of work relational database sample program has been shipped with the SQL product. There are several files and members within the QSQL library to help you set up an environment that will run a distributed DB2 UDB for iSeries sample program.

To use these files and members, you need to run the SETUP batch job located in the file QSQL/QSQSAMP. The SETUP batch job allows you to customize the example to do the following:

- Create the QSQSAMP library at the local and remote locations.
- Set up relational database directory entries at the local and remote locations.
- Create application panels at the local location.
- Precompile, compile, and run programs to create distributed sample application schemas, tables, indexes, and views.
- Load data into the tables at the local and remote locations.

- Precompile and compile programs.
- Create SQL packages at the remote location for the application programs.
- Precompile, compile, and run the program to update the location column in the department table.

Before running the SETUP, you may need to edit the SETUP member of the QSQL/QSQSAMP file. Instructions are included in the member as comments. To run the SETUP, specify the following command on the system command line:

```
=====> SBMDBJOB QSQL/QSQSAMP SETUP
```

Wait for the batch job to complete.

To use the sample program, specify the following command on the command line:

```
=====> ADDLIB QSQSAMP
```

To call the first display that allows you to customize the sample program, specify the following command on the command line.

```
=====> CALL QSQ8HC3
```

The following display appears. From this display, you can customize your database sample program.

```
DB2 for OS/400 ORGANIZATION APPLICATION

ACTION.....:  -   A (ADD)                E (ERASE)
D (DISPLAY)   -   U (UPDATE)

OBJECT.....:  -   DE (DEPARTMENT)         EM (EMPLOYEE)
DS (DEPT STRUCTURE)

SEARCH CRITERIA...:  -   DI (DEPARTMENT ID)   MN (MANAGER NAME)
DN (DEPARTMENT NAME)  -   EI (EMPLOYEE ID)
MI (MANAGER ID)      -   EN (EMPLOYEE NAME)

LOCATION.....:  _____ (BLANK IMPLIES LOCAL LOCATION)

DATA.....:  _____

                                           Bottom

F3=Exit

(C) COPYRIGHT IBM CORP. 1982, 1991
```

SQL package support

The OS/400 program supports an object called an SQL package. (OS/400 object type is *SQLPKG.) The SQL package contains the control structures and access plans necessary to process SQL statements on the application server when running a distributed program. An SQL package can be created when:

- The RDB parameter is specified on the CRTSQLxxx command and the program object is successfully created. The SQL package will be created on the system specified by the RDB parameter.

If the compile is unsuccessful or the compile only creates the module object, the SQL package will not be created.

- Using the CRTSQLPKG command. The CRTSQLPKG can be used to create a package when the package was not created at precompile time or if the package is needed at an RDB other than the one specified on the precompile command.

The Delete SQL Package (DLTSQLPKG) command allows you to delete an SQL package on the local system.

An SQL package is not created unless the privileges held by the authorization ID associated with the creation of the SQL package includes appropriate authority for creating a package on the remote system (the application server). To run the program, the authorization ID must include EXECUTE privileges on the SQL package. On iSeries systems, the EXECUTE privilege includes system authority of *OBJOPR and *EXECUTE.

The syntax for the Create SQL Package (CRTSQLPKG) command is shown in Appendix B, "DB2 UDB for iSeries CL Command Descriptions".

Valid SQL statements in an SQL package

Programs that connect to another server can use any of the SQL statements as described in the SQL Reference book, except the SET TRANSACTION statement. Programs compiled using DB2 UDB for iSeries that refer to a system that is not DB2 UDB for iSeries can use executable SQL statements supported by that remote system. The precompiler will continue to issue diagnostic messages for statements it does not understand. These statements are sent to the remote system during the creation of the SQL package. The run-time support will return a SQLCODE of -84 or -525 when the statement cannot be run on the current application server. For example, multiple-row FETCH, blocked INSERT, and scrollable cursor support are allowed only in distributed programs where both the application requester and application server are OS/400 at Version 2 Release 2 or later. For more information, see Considerations for Using Distributed Relational Database in the SQL Reference book.

Considerations for creating an SQL package

There are many considerations to think about when you are creating an SQL package. Some of these considerations are listed below.

CRTSQLPKG Authorization

When creating an SQL package on an iSeries system the authorization ID used must have *USE authority to the CRTSQLPKG command.

Creating a Package on a non-DB2 UDB for iSeries

When you create a program and SQL package for a non-DB2 UDB for iSeries, and try to use SQL statements that are unique to that relational database, the CRTSQLxxx GENLVL parameter should be set to 30. The program will be created unless a message with a severity level of greater than 30 is issued. If a message is issued with a severity level of greater than 30, the statement is probably not valid for any relational database. For example, undefined or unusable host variables or constants that are not valid would generate a message severity greater than 30.

The precompiler listing should be checked for unexpected messages when running with a GENLVL greater than 10. When you are creating a package for a DB2 Universal Database, you must set the GENLVL parameter to a value less than 20.

If the RDB parameter specifies a system that is not a DB2 UDB for iSeries system, then the following options should not be used on the CRTSQLxxx command:

- COMMIT(*NONE)
- OPTION(*SYS)
- DATFMT(*MDY)
- DATFMT(*DMY)
- DATFMT(*JUL)
- DATFMT(*YMD)
- DATFMT(*JOB)
- DYNUSRPRF(*OWNER)
- TIMFMT(*HMS) if TIMSEP(*BLANK) or TIMSEP(',')
- SRTSEQ(*JOB RUN)
- SRTSEQ(*LANGIDUNQ)
- SRTSEQ(*LANGIDSHR)
- SRTSEQ(library-name/table-name)

Note: When connecting to a DB2 Universal Database server, the following additional rules apply:

- The specified date and time formats must be the same format
- A value of *BLANK must be used for the TEXT parameter
- Default schemas (DFTRDBCOL) are not supported
- The CCSID of the source program from which the package is being created must not be 65535; if 65535 is used, an empty package is created.

Target Release (TGTRLS)

While creating the package, the SQL statements are checked to determine which release can support the function. This release is set as the restore level of the package. For example, if the package contains a CREATE TABLE statement which adds a FOREIGN KEY constraint to the table, then the restore level of the package will be Version 3 Release 1, because FOREIGN KEY constraints were not supported prior to this release. TGTRLS message are suppressed when the TGTRLS parameter is *CURRENT.

SQL Statement Size

The create SQL package function may not be able to handle the same size SQL statement that the precompiler can process. During the precompile of the SQL program, the SQL statement is placed into the associated space of the program. When this occurs, each token is separated by a blank. In addition, when the RDB parameter is specified, the host variables of the source statement are replaced with an 'H'. The create SQL package function passes this statement to the application server, along with a list of the host variables for that statement. The addition of the blanks between the tokens and the replacement of host variables may cause the statement to exceed the maximum SQL statement size (SQL0101 reason 5).

Statements that do not require a package

In some cases, you might try to create an SQL package but the SQL package will not be created and the program will still run. This situation occurs when the program contains only SQL statements that do not require an SQL package to run. For example, a program that contains only the SQL statement DESCRIBE TABLE will generate message SQL5041 during SQL package creation. The SQL statements that do not require an SQL package are:

- COMMIT
- CONNECT
- DESCRIBE TABLE

- DISCONNECT
- RELEASE
- RELEASE SAVEPOINT
- ROLLBACK
- SAVEPOINT
- SET CONNECTION

Package object type

SQL packages are always created as non-ILE objects and always run in the default activation group.

ILE programs and service programs

ILE programs and service programs that bind several modules containing SQL statements must have a separate SQL package for each module.

Package creation connection

The type of connection done for the package creation is based on the type of connect requested using the RDBCNNMTH parameter. If RDBCNNMTH(*DUW) was specified, commitment control is used and the connection may be a read-only connection. If the connection is read-only, then the package creation will fail.

Unit of work

Because package creation implicitly performs a commit or rollback, the commit definition must be at a unit of work boundary before the package creation is attempted. The following conditions must all be true for a commit definition to be at a unit of work boundary:

- SQL is at a unit of work boundary.
- There are no local or DDM files open using commitment control and no closed local or DDM files with pending changes.
- There are no API resources registered.
- There are no LU 6.2 resources registered that are not associated with DRDA or DDM.

Creating packages locally

The name specified on the RDB parameter can be the name of the local system. If it is the name of the local system, the SQL package will be created on the local system. The SQL package can be saved (SAVOBJ command) and then restored (RSTOBJ command) to another server. When you run the program with a connection to the local system, the SQL package is not used. If you specify *LOCAL for the RDB parameter, an *SQLPKG object is not created, but the package information is saved in the *PGM object.

Labels

You can use the LABEL ON statement to create a description for the SQL package.

Consistency token

The program and its associated SQL package contain a consistency token that is checked when a call is made to the SQL package. The consistency tokens must match or the package cannot be used. It is possible for the program and SQL package to appear to be uncoordinated. Assume the program is on the iSeries system and the application server is another iSeries system. The program is running in session A and it is recreated in session B (where the SQL package is also recreated). The next call to the program in session A could result in a consistency token error. To avoid locating the SQL package on each call, SQL maintains a list of addresses for SQL packages that are used by each session. When

session B re-creates the SQL package, the old SQL package is moved to the QRPLOBJ library. The address to the SQL package in session A is still valid. (This situation can be avoided by creating the program and SQL package from the session that is running the program, or by submitting a remote command to delete the old SQL package before creating the program.)

To use the new SQL package, you should end the connection with the remote system. You can either sign off the session and then sign on again, or you can use the interactive SQL (STRSQL) command to issue a DISCONNECT for unprotected network connections or a RELEASE followed by a COMMIT for protected connections. RCLDDMCNV should then be used to end the network connections. Call the program again.

SQL and recursion

If you invoke SQL from an attention key program while you are already precompiling, you will receive unpredictable results.

The CRTSQLxxx, CRTSQLPKG, STRSQL commands and the SQL run-time environment are not recursive. They will produce unpredictable results if recursion is attempted. Recursion would occur if while one of the commands is running, (or running a program with embedded SQL statements) the job is interrupted before the command has completed, and another SQL function is started.

CCSID considerations for SQL

If you are running a distributed application and one of your systems is not an iSeries system, the job CCSID value on the iSeries server cannot be set to 65535.

Before requesting that the remote system create an SQL package, the application requester always converts the name specified on the RDB parameter, SQL package name, library name, and the text of the SQL package from the CCSID of the job to CCSID 500. This is required by DRDA. When the remote relational database is an iSeries system, the names are not converted from CCSID 500 to the job CCSID.

It is recommended that delimited identifiers not be used for table, view, index, schema, library, or SQL package names. Conversion of names does not occur between systems with different CCSIDs. Consider the following example with system A running with a CCSID of 37 and system B running with a CCSID of 500.

- Create a program that creates a table with the name "a¬b|c" on system A.
- Save program "a¬b|c" on system A, then restore it to system B.
- The code point for ¬ in CCSID 37 is x'5F' while in CCSID 500 it is x'BA'.
- On system B the name would display "a[b]c". If you created a program that referenced the table whose name was "a¬b|c.", the program would not find the table.

The at sign (@), pound sign (#), and dollar sign (\$) characters should not be used in SQL object names. Their code points depend on the CCSID used. If you use delimited names or the three national extenders, the name resolution functions may possibly fail in a future release.

Connection management and activation groups

For details, see the following topics:

- “Connections and conversations”
- “Source code for PGM1:” on page 335
- “Source code for PGM2:” on page 335
- “Source code for PGM3:” on page 335
- “Multiple connections to the same relational database” on page 337
- “Implicit connection management for the default activation group” on page 338
- “Implicit connection management for nondefault activation groups” on page 339

Connections and conversations

Prior to the use of TCP/IP by DRDA, the term ‘connection’ was not ambiguous. It referred to a connection from the SQL point of view. That is, a connection started at the time one did a `CONNECT TO` some RDB, and ended when a `DISCONNECT` was done or a `RELEASE ALL` followed by a successful `COMMIT` occurred. The APPC conversation may or may not have been kept up, depending on the job’s `DDMCNV` attribute value, and whether the conversation was with an iSeries or other type of system.

TCP/IP terminology does not include the term ‘conversation’. A similar concept exists, however. With the advent of TCP/IP support by DRDA, use of the term ‘conversation’ will be replaced, in this book, by the more general term ‘connection’, unless the discussion is specifically about an APPC conversation. Therefore, there are now two different types of connections about which the reader must be aware: SQL connections of the type described above, and ‘network’ connections which replace the term ‘conversation’.

Where there would be the possibility of confusion between the two types of connections, the word will be qualified by ‘SQL’ or ‘network’ to allow the reader to understand the intended meaning.

SQL connections are managed at the activation group level. Each activation group within a job manages its own connections and these connections are not shared across activation groups. For programs that run in the default activation group, connections are still managed as they were prior to Version 2 Release 3.

The following is an example of an application that runs in multiple activation groups. This example is used to illustrate the interaction between activation groups, connection management, and commitment control. It is **not** a recommended coding style.

Source code for PGM1:

```
....  
EXEC SQL  
    CONNECT TO SYSB  
END-EXEC.  
EXEC SQL  
    SELECT ....  
END-EXEC.  
CALL PGM2.  
....
```

Figure 9. Source Code for PGM1

Command to create program and SQL package for PGM1:
CRTSQLCBL PGM(PGM1) COMMIT(*NONE) RDB(SYSB)

Source code for PGM2:

```
...  
EXEC SQL  
    CONNECT TO SYSC;  
EXEC SQL  
    DECLARE C1 CURSOR FOR  
        SELECT ....;  
EXEC SQL  
    OPEN C1;  
do {  
    EXEC SQL  
        FETCH C1 INTO :st1;  
    EXEC SQL  
        UPDATE ...  
            SET COL1 = COL1+10  
            WHERE CURRENT OF C1;  
    PGM3(st1);  
} while SQLCODE == 0;  
EXEC SQL  
    CLOSE C1;  
EXEC SQL COMMIT;  
....
```

Figure 10. Source Code for PGM2

Command to create program and SQL package for PGM2:
CRTSQLCI OBJ(PGM2) COMMIT(*CHG) RDB(SYSC) OBJTYPE(*PGM)

Source code for PGM3:

```
...  
EXEC SQL  
    INSERT INTO TAB VALUES(:st1);  
EXEC SQL COMMIT;  
....
```

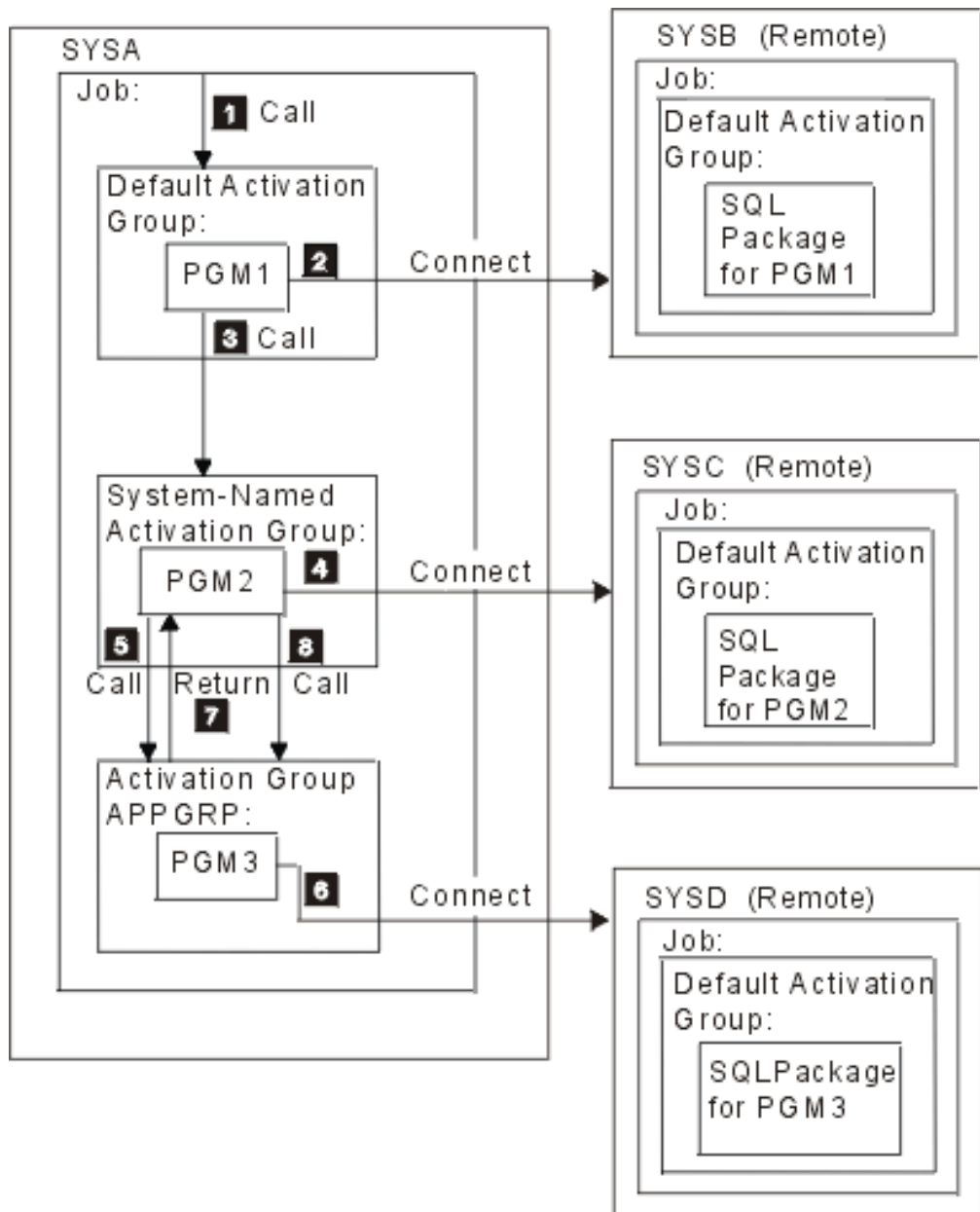
Figure 11. Source Code for PGM3

Commands to create program and SQL package for PGM3:

```

CRTSQLCI OBJ(PGM3) COMMIT(*CHG) RDB(SYSD) OBJTYPE(*MODULE)
CRTPGM PGM(PGM3) ACTGRP(APPGRP)
CRTSQLPKG PGM(PGM3) RDB(SYSD)

```



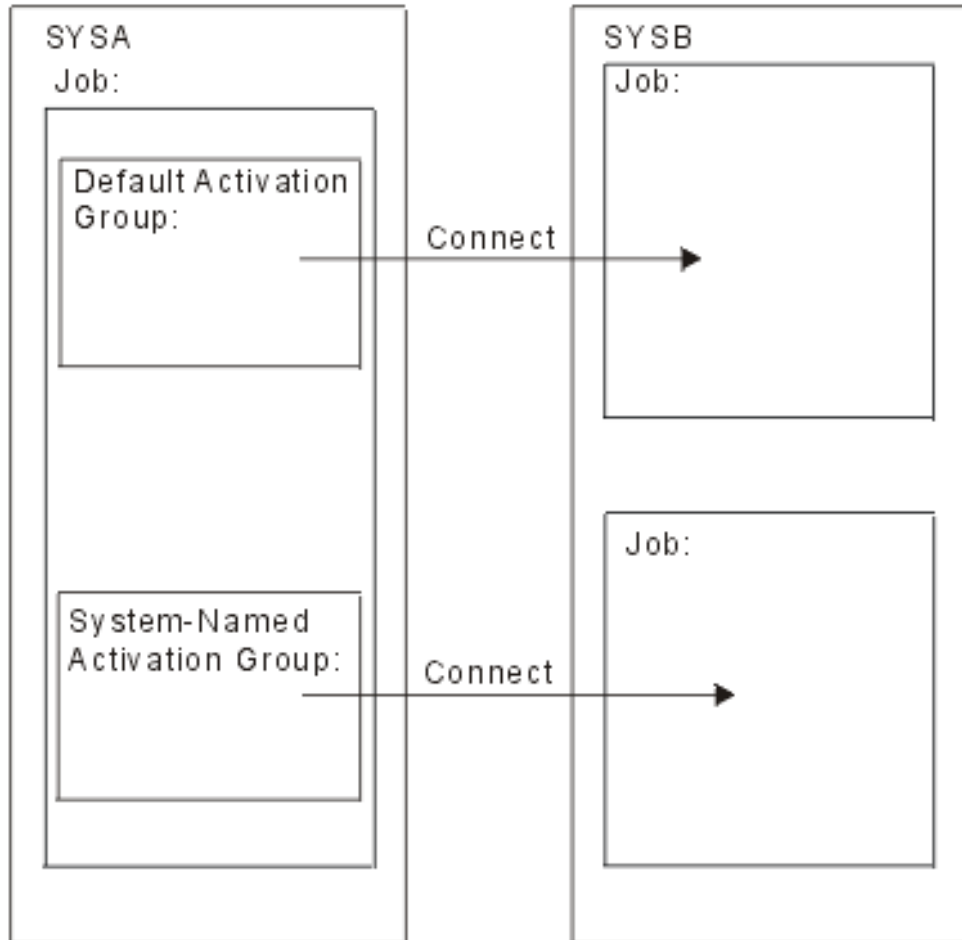
In this example, PGM1 is a non-ILE program created using the CRTSQLCBL command. This program runs in the default activation group. PGM2 is created using the CRTSQLCI command, and it runs in a system-named activation group. PGM3 is also created using the CRTSQLCI command, but it runs in the activation group named APPGRP. Because APPGRP is not the default value for the ACTGRP parameter, the CRTPGM command is issued separately. The CRTPGM command is followed by a CRTSQLPKG command that creates the SQL package object on the SYSD relational database. In this example, the user has not explicitly started the job level commitment definition. SQL implicitly starts commitment control.

1. PGM1 is called and runs in the default activation group.

2. PGM1 connects to relational database SYSB and runs a SELECT statement.
3. PGM1 then calls PGM2, which runs in a system-named activation group.
4. PGM2 does a connect to relational database SYSC. Because PGM1 and PGM2 are in different activation groups, the connection started by PGM2 in the system-named activation group does not disconnect the connection started by PGM1 in the default activation group. Both connections are active. PGM2 opens the cursor and fetches and updates a row. PGM2 is running under commitment control, is in the middle of a unit of work, and is not at a connectable state.
5. PGM2 calls PGM3, which runs in activation group APPGRP.
6. The INSERT statement is the first statement run in activation group APPGRP. The first SQL statement causes an implicit connect to relational database SYSD. A row is inserted into table TAB located at relational database SYSD. The insert is then committed. The pending changes in the system-named activation group are not committed, because commitment control was started by SQL with a commit scope of activation group.
7. PGM3 is then exited and control returns to PGM2. PGM2 fetches and updates another row.
8. PGM3 is called again to insert the row. An implicit connect was done on the first call to PGM3. It is not done on subsequent calls because the activation group did not end between calls to PGM3. Finally, all the rows are processed by PGM2 and the unit of work associated with the system-named activation group is committed.

Multiple connections to the same relational database

If different activation groups connect to the same relational database, each SQL connection has its own network connection and its own application server job. If activation groups are run with commitment control, changes committed in one activation group do not commit changes in other activation groups unless the job-level commitment definition is used.



Implicit connection management for the default activation group

The application requester can implicitly connect to an application server. Implicit SQL connection occurs when the application requester detects the first SQL statement is being issued by the first active SQL program for the default activation group and the following items are true:

- The SQL statement being issued is not a CONNECT statement with parameters.
- SQL is not active in the default activation group.

For a distributed program, the implicit SQL connection is to the relational database specified on the RDB parameter. For a nondistributed program, the implicit SQL connection is to the local relational database.

SQL will end any active connections in the default activation group when SQL becomes not active. SQL becomes not active when:

- The application requester detects the first active SQL program for the process has ended and the following are all true:
 - There are no pending SQL changes
 - There are no connections using protected connections

- A SET TRANSACTION statement is not active
- No programs that were precompiled with CLOSQLCSR(*ENDJOB) were run.

If there are pending changes, protected connections, or an active SET TRANSACTION statement, SQL is placed in the exited state. If programs precompiled with CLOSQLCSR(*ENDJOB) were run, SQL will remain active for the default activation group until the job ends.

- At the end of a unit of work, if SQL is in the exited state. This occurs when you issue a COMMIT or ROLLBACK command outside of an SQL program.
- At the end of a job.

Implicit connection management for nondefault activation groups

The application requester can implicitly connect to an application server. Implicit SQL connection occurs when the application requester detects that the first SQL statement issued for the activation group is not a CONNECT statement with parameters.

For a distributed program, the implicit SQL connection is made to the relational database specified on the RDB parameter. For a nondistributed program, the implicit SQL connection is made to the local relational database.

Implicit disconnect can occur at the following times in a process:

- When the activation group ends, if commitment control is not active, activation group level commitment control is active, or the job level commitment definition is at a unit of work boundary.
 - If the job level commitment definition is active and not at a unit of work boundary, SQL is placed in the exited state.
- If SQL is in the exited state, when the job level commitment definition is committed or rolled back.
- At the end of a job.

Distributed support

DB2 UDB for iSeries supports two levels of distributed relational database:

- Remote unit of work (RUW)

Remote unit of work is where the preparation and running of SQL statements occurs at only one application server during a unit of work. An activation group with an application process at an application requester can connect to an application server and, within one or more units of work, run any number of static or dynamic SQL statements that refer to objects on the application server. Remote unit of work is also referred to as DRDA level 1.

- Distributed unit of work (DUW)

Distributed unit of work is where the preparation and running of SQL statements can occur at multiple applications servers during a unit of work. However, a single SQL statement can only refer to objects located at a single application server. Distributed unit of work is also referred to as DRDA level 2.

Distributed unit of work allows:

- Update access to multiple application servers in one logical unit of work
or

- Update access to a single application server with read access to multiple application servers, in one logical unit of work.

Whether multiple application servers can be updated in a unit of work is dependent on the existence of a sync point manager at the application requester, sync point managers at the application servers, and two-phase commit protocol support between the application requester and the application servers.

The sync point manager is a system component that coordinates commit and rollback operations among the participants in the two-phase commit protocol. When running distributed updates, the sync point managers on the different systems cooperate to ensure that resources reach a consistent state. The protocols and flows used by sync point managers are also referred to as two-phase commit protocols.

If two-phase commit protocols will be used, the connection is a protected resource; otherwise the connection is an unprotected resource.

The type of data transport protocols used between systems affects whether the network connection is protected or unprotected. Before V5R1, TCP/IP connections were always unprotected; thus they could participate in a distributed unit of work in only a limited way. In V5R1, full support for DUW with TCP/IP was added.

For example, if the first connection made from the program is to a pre-V5R1 server over TCP/IP, updates can be performed over it, but any subsequent connections, even over APPC, will be read only.

Note that when using Interactive SQL, the first SQL connection is to the local system. Therefore, in the pre-V5R1 environment, in order to make updates to a remote system using TCP/IP, you must do a `RELEASE ALL` followed by a `COMMIT` to end all SQL connections before doing the `CONNECT TO remote-tcp-system`.

Determining connection type

When a remote connection is established it will use either an unprotected or protected network connection. With regards to committable updates, this SQL connection may be read-only, updateable, or unknown whether it is updateable when the connection is established. A committable update is any insert, delete, update, or DDL statement that is run under commitment control. If the connection is read-only, changes using `COMMIT(*NONE)` can still be run. After a `CONNECT` or `SET CONNECTION`, `SQLERRD(4)` of the `SQLCA` indicates the type of connection. `SQLERRD(4)` will also indicate if the connection uses a unprotected or protected network connection. Specific values are:

1. Committable updates can be performed on the connection. The connection is unprotected. This will occur when:
 - The connection is established using remote unit of work (`RDBCNNMTH(*RUW)`). This also includes local connections and application requester driver (ARD) connections using remote unit of work.
 - If the connection is established using distributed unit of work (`RDBCNNMTH(*DUW)`) then all the following are true:
 - The connection is not local.

- The application server does not support distributed unit of work. For example, a DB2 UDB for iSeries application server with a release of OS/400 prior to Version 3 Release 1.
- The commitment control level of the program issuing the connect is not *NONE.
- Either no connections to other application servers (including local) exist that can perform committable updates or all connections are read-only connections to application servers that do not support distributed unit of work.
- There are no open updateable local files under commitment control for the commitment definition.
- There are no open updateable DDM files that use a different connection under commitment control for the commitment definition.
- There are no API commitment control resources for the commitment definition.
- There are no protected connections registered for the commitment definition.

If running with commitment control, SQL will register a one-phase updateable DRDA resource for remote connections or a two-phase updateable DRDA resource for local and ARD connections.

2. No committable updates can be performed on the connection. The connection is read-only. The network connection is unprotected.

This will never occur for applications compiled with remote unit of work connection management (*RUW).

For distributed unit of work applications, this will occur only when the following are true when the connection is established:

- The connection is not local.
- The application server does not support distributed unit of work
- At least one of the following is true:
 - The commitment control level of the program issuing the connect is *NONE.
 - Another connection exists to an application server that does not support distributed unit-of-work and that application server can perform committable updates
 - Another connection exists to an application server that supports distributed unit-of-work (including local).
 - There are open updateable local files under commitment control for the commitment definition.
 - There are open updateable DDM files that use a different connection under commitment control for the commitment definition.
 - There are no one-phase API commitment control resources for the commitment definition.
 - There are protected connections registered for the commitment definition.

If running with commitment control, SQL will register a one-phase read-only resource.

3. It is unknown if committable updates can be performed. The connection is protected.

This will never occur for applications compiled with remote unit of work connection management (*RUW).

For distributed unit of work applications, this will occur when all of the following are true when the connection is established:

- The connection is not local.
- The commitment control level of the program issuing the connect is not *NONE.
- The application server supports both distributed unit of work and two-phase commit protocol (protected connections).

If running with commitment control, SQL will register a two-phase undetermined resource.

4. It is unknown if committable updates can be performed. The connection is not protected.

This will never occur for applications compiled with remote unit of work connection management (*RUW).

For distributed unit of work, this will occur only when all of the following are true when the connection is established:

- The connection is not local.
- The application server supports distributed unit of work
- Either the application server does not support two-phase commit protocols (protected connections) or the commitment control level of the program issuing the connect is *NONE.

If running with commitment control, SQL will register a one-phase DRDA undetermined resource.

5. It is unknown if committable updates can be performed and the connection is a local connection using distributed unit of work or an ARD connection using distributed unit of work.

If running with commitment control, SQL will register a two-phase DRDA undetermined resource.

For more information about two-phase and one-phase resources, see the

Commitment control topic.Backup and Recovery  book.

The following table summarizes the type of connection that will result for remote distributed unit of work connections. SQLERRD(4) is set on successful CONNECT and SET CONNECTION statements.

Table 39. Summary of Connection Type

Connect under Commitment Control	Application Server Supports Two-phase Commit	Application Server Supports Distributed Unit of Work	Other Updateable One-phase Resource Registered	SQLERRD(4)
No	No	No	No	2
No	No	No	Yes	2
No	No	Yes	No	4
No	No	Yes	Yes	4
No	Yes	No	No	2
No	Yes	No	Yes	2
No	Yes	Yes	No	4

Table 39. Summary of Connection Type (continued)

Connect under Commitment Control	Application Server Supports Two-phase Commit	Application Server Supports Distributed Unit of Work	Other Updateable One-phase Resource Registered	SQLERRD(4)
No	Yes	Yes	Yes	4
Yes	No	No	No	1
Yes	No	No	Yes	2
Yes	No	Yes	No	4
Yes	No	Yes	Yes	4
Yes	Yes	No	No	N/A *
Yes	Yes	No	Yes	N/A *
Yes	Yes	Yes	No	3
Yes	Yes	Yes	Yes	3

*DRDA does not allow protected connections to be used to application servers which only support remote unit of work (DRDA1). This includes all DB2 for iSeries TCP/IP connections.

Connect and commitment control restrictions

There are some restrictions on when you can connect using commitment control. These restrictions also apply to attempting to run statements using commitment control but the connection was established using COMMIT(*NONE).

If a two-phase undetermined or updateable resource is registered or a one-phase updateable resource is registered, another one-phase updateable resource cannot not be registered.

Furthermore, when protected connections are inactive and the DDMCNV job attribute is *KEEP, these unused DDM connections will also cause the CONNECT statements in programs compiled with RUW connection management to fail.

If running with RUW connection management and using the job-level commitment definition, then there are some restrictions.

- If the job-level commitment definition is used by more than one activation group, all RUW connections must be to the local relational database.
- If the connection is remote, only one activation group may use the job-level commitment definition for RUW connections.

Determining connection status

The CONNECT statement without parameters can be used to determine if the current connection is updateable or read-only for the current unit of work. A value of 1 or 2 will be returned in SQLERRD(3). The value in SQLERRD(3) is determined as follows:

1. Committable updates can be performed on the connection for the unit of work. This will occur when one of the following is true:
 - SQLERRD(4) has a value of 1.
 - All of the following are true:
 - SQLERRD(4) has a value of 3 or 5.

- No connection exists to an application server that does not support distributed unit of work which can perform committable updates.
 - One of the following is true:
 - The first committable update is performed on a connection that uses a protected connection, is performed on the local database, or is performed on a connection to an ARD program.
 - There there are open updateable local files under commitment control.
 - There are open updateable DDM files that use protected connections.
 - There are two-phase API commitment control resources.
 - No committable updates have been made.
 - All of the following are true:
 - SQLERRD(4) has a value of 4
 - No other connections exist to an application server that does not support distributed unit of work which can perform committable updates.
 - The first committable update is performed on this connection or no committable updates have been made.
 - There are no open updateable DDM files that use protected connections.
 - There are no open updateable local files under commitment control.
 - There are no two-phase API commitment control resources.
2. No committable updates can be performed on the connection for this unit of work.

This will occur when one of the following is true:

- SQLERRD(4) has a value of 2.
- SQLERRD(4) has a value of 3 or 5 and one of the following is true:
 - A connection exists to an updateable application server that only supports remote unit of work.
 - The first committable update is performed on a connection that uses an unprotected connection.
- SQLERRD(4) has a value of 4 and one of the following is true:
 - A connection exists to an updateable application server that only supports remote unit of work.
 - The first committable update was not performed on this connection.
 - There are open updateable DDM files that use protected connections.
 - There are open updateable local files under commitment control.
 - There are two-phase API commitment control resources.

This following table summarizes how SQLERRD(3) is determined based on the SQLERRD(4) value, if there is an updateable connection to an application server that only supports remote unit of work, and where the first committable update occurred.

Table 40. Summary of Determining SQLERRD(3) Values

SQLERRD(4)	Connection Exists to Updateable Remote Unit of Work Application Server	Where First Committable Update Occurred *	SQLERRD(3)
1	--	--	1
2	--	--	2
3	Yes	--	2
3	No	no updates	1
3	No	one-phase	2
3	No	this connection	1
3	No	two-phase	1
4	Yes	--	2
4	No	no updates	1
4	No	one-phase	2
4	No	this connection	1
4	No	two-phase	2
5	Yes	--	2
5	No	no updates	1
5	No	one-phase	2
5	No	this connection	1
5	No	two-phase	1

* The terms in this column are defined as:

- *No updates* indicates no committable updates have been performed, no DDM files open for update using a protected connection, no local files are open for update, and no commitment control APIs are registered.
- *One-phase* indicates the first committable update was performed using an unprotected connection or DDM files are open for update using unprotected connections.
- *Two-phase* indicates a committable update was performed on a two-phase distributed-unit-of-work application server, DDM files are open for update using a protected connection, commitment control APIs are registered, or local files are open for update under commitment control.

When the value of SQLERRD(4) is 3, 4, or 5 (due to an ARD program) and the value of SQLERRD(3) is 2, if an attempt is made to perform a committable update over the connection, the unit of work will be placed in a rollback required state. If an unit of work is in a rollback required state, the only statement allowed is a ROLLBACK statement; all other statements will result in SQLCODE -918.

Distributed unit of work connection considerations

When connecting in a distributed unit of work application, there are many considerations. This section lists some design considerations.

- If the unit of work will perform updates at more than one application server and commitment control will be used, all connections over which updates will be done should be made using commitment control. If the connections are done not using commitment control and later committable updates are performed, read-only connections for the unit of work are likely to result.

- Other non-SQL commit resources, such as local files, DDM files, and commitment control API resources, will affect the updateable and read-only status of a connection.
- If connecting using commitment control to an application server that does not support distributed unit of work (for example, a V4R5 iSeries using TCP/IP), that connection will be either updateable or read-only. If the connection is updateable it is the only updateable connection.

Ending connections

Because remote connections use resources, connections that are no longer going to be used should be ended as soon as possible. Connections can be ended implicitly or explicitly. For a description of when connections are implicitly ended see “Implicit connection management for the default activation group” on page 338 and “Implicit connection management for nondefault activation groups” on page 339. Connections can be explicitly ended by either the DISCONNECT statement or the RELEASE statement followed by a successful COMMIT. The DISCONNECT statement can only be used with connections that use unprotected connections or with local connections. The DISCONNECT statement will end the connection when the statement is run. The RELEASE statement can be used with either protected or unprotected connections. When the RELEASE statement is run, the connection is not ended but instead placed into the released state. A connection that is in the release state can still be used. The connection is not ended until a successful COMMIT is run. A ROLLBACK or an unsuccessful COMMIT will not end a connection in the released state.

When a remote SQL connection is established, a DDM network connection (APPC conversation or TCP/IP connection) is used. When the SQL connection is ended, the network connection may either be placed in the unused state or dropped. Whether a network connection is dropped or placed in the unused state depends on the DDMCNV job attribute. If the job attribute value is *KEEP and the connection is to another iSeries server, the connection becomes unused. If the job attribute value is *DROP and the connection is to another iSeries server, the connection is dropped. If the connection is to a non-iSeries server, the connection is always dropped. *DROP is desirable in the following situations:

- When the cost of maintaining the unused connection is high and the connection will not be used relatively soon.
- When running with a mixture of programs, some compiled with RUW connection management and some programs compiled with DUW connection management. Attempts to run programs compiled with RUW connection management to remote locations will fail when protected connections exist.
- When running with protected connections using either DDM or DRDA. Additional overhead is incurred on commits and rollbacks for unused protected connections.

The Reclaim DDM connections (RCLDDMCNV) command may be used to end all unused connections, if they are at a commit boundary.

Distributed unit of work

Distributed unit of work (DUW) allows access to multiple application servers within the same unit of work. Each SQL statement can access only one application server. Using distributed unit of work allows changes at multiple applications servers to be committed or rolled back within a single unit of work.

Managing distributed unit of work connections

The CONNECT, SET CONNECTION, DISCONNECT, and RELEASE statements are used to manage connections in the DUW environment. A distributed unit of work CONNECT is run when the program is precompiled using RDBCNNMTH(*DUW), which is the default. This form of the CONNECT statement does not disconnect existing connections but instead places the previous connection in the dormant state. The relational database specified on the CONNECT statement becomes the current connection. The CONNECT statement can only be used to start new connections; if you want to switch between existing connections, the SET CONNECTION statement must be used. Because connections use system resources, connections should be ended when they are no longer needed. The RELEASE or DISCONNECT statement can be used to end connections. The RELEASE statement must be followed by a successful commit in order for the connections to end.

The following is an example of a C program running in a DUW environment that uses commitment control.

```
....
EXEC SQL WHENEVER SQLERROR GO TO done;
EXEC SQL WHENEVER NOT FOUND GO TO done;
....
EXEC SQL
  DECLARE C1 CURSOR WITH HOLD FOR
  SELECT PARTNO, PRICE
  FROM PARTS
  WHERE SITES_UPDATED = 'N'
  FOR UPDATE OF SITES_UPDATED;
/* Connect to the systems */
EXEC SQL CONNECT TO LOCALSYS;
EXEC SQL CONNECT TO SYSB;
EXEC SQL CONNECT TO SYSC;
/* Make the local system the current connection */
EXEC SQL SET CONNECTION LOCALSYS;
/* Open the cursor */
EXEC SQL OPEN C1;
```

Figure 12. Example of Distributed Unit of Work Program (Part 1 of 4)

```
while (SQLCODE==0)
{
  /* Fetch the first row */
  EXEC SQL FETCH C1 INTO :partnumber,:price;
  /* Update the row which indicates that the updates have been
  propagated to the other sites */
  EXEC SQL UPDATE PARTS SET SITES_UPDATED='Y'
  WHERE CURRENT OF C1;
  /* Check if the part data is on SYSB */
  if ((partnumber > 10) && (partnumber < 100))
  {
    /* Make SYSB the current connection and update the price */
    EXEC SQL SET CONNECTION SYSB;
    EXEC SQL UPDATE PARTS
    SET PRICE=:price
    WHERE PARTNO=:partnumber;
  }
}
```

Figure 12. Example of Distributed Unit of Work Program (Part 2 of 4)

```

/* Check if the part data is on SYSC */
if ((partnumber > 50) && (partnumber < 200))
{
    /* Make SYSC the current connection and update the price */
    EXEC SQL SET CONNECTION SYSC;
    EXEC SQL UPDATE PARTS
        SET PRICE=:price
        WHERE PARTNO=:partnumber;
}
/* Commit the changes made at all 3 sites */
EXEC SQL COMMIT;
/* Set the current connection to local so the next row
can be fetched */
EXEC SQL SET CONNECTION LOCALSYS;
}
done:

```

Figure 12. Example of Distributed Unit of Work Program (Part 3 of 4)

```

EXEC SQL WHENEVER SQLERROR CONTINUE;
/* Release the connections that are no longer being used */
EXEC SQL RELEASE SYSB;
EXEC SQL RELEASE SYSC;
/* Close the cursor */
EXEC SQL CLOSE C1;
/* Do another commit which will end the released connections.
The local connection is still active because it was not
released. */
EXEC SQL COMMIT;
...

```

Figure 12. Example of Distributed Unit of Work Program (Part 4 of 4)

In this program, there are 3 application servers active: LOCALSYS which the local system, and 2 remote systems, SYSB and SYSC. SYSB and SYSC also support distributed unit of work and two-phase commit. Initially all connections are made active by using the CONNECT statement for each of the application servers involved in the transaction. When using DUW, a CONNECT statement does not disconnect the previous connection, but instead places the previous connection in the dormant state. After all the application servers, have been connected, the local connection is made the current connection using the SET CONNECTION statement. The cursor is then opened and the first row of data fetched. It is then determined at which application servers the data needs to be updated. If SYSB needs to be updated, then SYSB is made the current connection using the SET CONNECTION statement and the update is run. The same is done for SYSC. The changes are then committed. Because two-phase commit is being used, it is guaranteed that the changes are committed at the local system and the two remote systems. Because the cursor was declared WITH HOLD, it remains open after the commit. The current connection is then changed to the local system so that the next row of data can be fetched. This set of fetches, updates, and commits is repeated until all the data has been processed. After all the data has been fetched, the connections for both remote systems are released. They can not be disconnected because they use protected connections. After the connections are released, a commit is issued to end the connections. The local system is still connected and continues processing.

Checking connection status

If running in an environment where it is possible to have read-only connections, the status of the connection should be checked before doing committable updates.

This will prevent the unit of work from entering the rollback required state. The following COBOL example shows how to check the connection status.

```
...
EXEC SQL
  SET CONNECTION SYS5
END-EXEC.
...
* Check if the connection is updateable.
EXEC SQL CONNECT END-EXEC.
* If connection is updateable, update sales information otherwise
* inform the user.
IF SQLERRD(3) = 1 THEN
EXEC SQL
  INSERT INTO SALES_TABLE
  VALUES(:SALES-DATA)
END-EXEC
ELSE
  DISPLAY 'Unable to update sales information at this time'.
...

```

Figure 13. Example of Checking Connection Status

Cursors and prepared statements

Cursors and prepared statements are scoped to the compilation unit and also to the connection. Scoping to the compilation unit means that a program called from another separately compiled program cannot use a cursor or prepared statement that was opened or prepared by the calling program. Scoping to the connection means that each connection within a program can have its own separate instance of a cursor or prepared statement.

The following distributed unit of work example shows how the same cursor name is opened in two different connections, resulting in two instances of cursor C1.

```

.....
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT * FROM CORPDATA.EMPLOYEE;
/* Connect to local and open C1 */
EXEC SQL CONNECT TO LOCALSYS;
EXEC SQL OPEN C1;
/* Connect to the remote system and open C1 */
EXEC SQL CONNECT TO SYSA;
EXEC SQL OPEN C1;
/* Keep processing until done */
while (NOT_DONE) {
    /* Fetch a row of data from the local system */
    EXEC SQL SET CONNECTION LOCALSYS;
    EXEC SQL FETCH C1 INTO :local_emp_struct;
    /* Fetch a row of data from the remote system */
    EXEC SQL SET CONNECTION SYSA;
    EXEC SQL FETCH C1 INTO :rmt_emp_struct;
    /* Process the data */
    .....
}
/* Close the cursor on the remote system */
EXEC SQL CLOSE C1;
/* Close the cursor on the local system */
EXEC SQL SET CONNECTION LOCALSYS;
EXEC SQL CLOSE C1;
.....

```

Figure 14. Example of Cursors in a DUW program

Application requester driver programs

To complement database access provided by products that implement DRDA, DB2 UDB for iSeries provides an interface for writing exit programs on a DB2 UDB for iSeries application requester to process SQL requests. Such an exit program is called an **application requester driver**. The server calls the ARD program during the following operations:

- During package creation performed using the CRTSQLPKG or CRTSQLxxx commands, when the relational database (RDB) parameter matches the RDB name corresponding to the ARD program.
- Processing of SQL statements when the current connection is to an RDB name corresponding to the ARD program.

These calls allow the ARD program to pass the SQL statements and information about the statements to a remote relational database and return results back to the system. The system then returns the results to the application or the user. Access to relational databases accessed by ARD programs appears like access to DRDA application servers in the unlike environment. However, not all DRDA function is supported in the ARD environment. Examples of function not supported are Large objects (LOBs) and long passwords (passphrases).

For more information about application requester driver programs, see the OS/400 File APIs.

Problem handling

The primary strategy for capturing and reporting error information for the distributed database function is called **first failure data capture (FFDC)**. The purpose of FFDC support is to provide accurate information on errors detected in the DDM components of the OS/400 system from which an APAR (Authorized Program Analysis Report) can be created. By means of this function, key structures and the DDM data stream are automatically dumped to a spool file. The first 1024 bytes of the error information are also logged in the system error log. This automatic dumping of error information on the first occurrence of an error means that the failure should not have to be recreated to be reported by the customer. FFDC is active in both the application requester and application server functions of the OS/400 DDM component. However, for the FFDC data to be logged, the system value QSFWERRLOG must be set to *LOG.

Note: Not all negative SQLCODEs are dumped; only those that can be used to produce an APAR are dumped. For more information about handling problems on distributed relational database operations, see the *Distributed Database Problem Determination Guide*

When an SQL error is detected, an SQLCODE with a corresponding SQLSTATE is returned in the SQLCA. For more information about these codes, see the SQL messages and codes topic in the iSeries Information Center.

DRDA stored procedure considerations

The iSeries DRDA server supports the return of one or more result sets from a stored procedure. Note, however, that in V5R1, only server enablement is provided, so that the feature can be used only from a non-iSeries client that supports stored procedure result sets.

In V5R2, iSeries client-side support was added for applications that use the CLI interface for SQL. However, you must apply a PTF to enable V5R1 iSeries servers to return stored procedure result sets to V5R2 iSeries clients. See the PTF Cover

Letter Database  to see a list of cover letters sorted by release, by date, or by fix number.

Result sets can be generated in the stored procedure by opening one or more SQL cursors associated with SQL SELECT statements. In addition, a maximum of one array result set can also be returned. For more information about writing stored procedures that return result sets, see the descriptions of the SET RESULT SETS, CREATE PROCEDURE (SQL), and CREATE PROCEDURE (External) statements in the SQL Reference book. For general information about the use of stored procedures with DRDA, see the Distributed Database Programming book.

Appendix A. DB2 UDB for iSeries Sample Tables

This appendix contains the sample tables referred to and used in this guide and the SQL Reference book. Along with the tables are the SQL statements for creating the tables. For detailed information on creating tables, see “Creating and using a table” on page 16.

As a group, the tables include information that describes employees, departments, projects, and activities. This information makes up a sample application demonstrating some of the features of the DB2 UDB Query Manager and SQL Development Kit licensed program. All examples assume the tables are in a schema named CORPDATA (for corporate data).

A stored procedure is shipped as part of the system that contains the DDL statements to create all of these tables, and the INSERT statements to populate them. The procedure will create the schema specified on the call to the procedure. Since this is an SQL external stored procedure, it can be called from any SQL interface, including interactive SQL and iSeries Navigator. To invoke the procedure where *SAMPLE* is the schema you wish to create, issue the following statement:

```
CALL QSYS.CREATE_SQL_SAMPLE ('SAMPLE')
```

The schema name must be specified in uppercase. The schema must not already exist.

The tables are:

- “Department Table (DEPARTMENT)” on page 354
- “Employee Table (EMPLOYEE)” on page 355
- “Employee Photo Table (EMP_PHOTO)” on page 357
- “Employee Resume Table (EMP_RESUME)” on page 358
- “Employee to Project Activity Table (EMPPROJECT)” on page 359
- “Project Table (PROJECT)” on page 362
- “Project Activity Table (PROJACT)” on page 364
- “Activity Table (ACT)” on page 366
- “Class Schedule Table (CL_SCHED)” on page 367
- “In Tray Table (IN_TRAY)” on page 368

Indexes, aliases, and views are created for many of these tables. The view definitions are not included here.

There are three other tables created as well that are not related to the first set.

- “Organization Table (ORG)” on page 369
- “Staff Table (STAFF)” on page 370
- “Sales Table (SALES)” on page 371

Notes:

1. In these sample tables, a question mark (?) indicates a null value.

Department Table (DEPARTMENT)

The department table describes each department in the enterprise and identifies its manager and the department it reports to. The department table is created with the following CREATE TABLE and ALTER TABLE statements:

```
CREATE TABLE DEPARTMENT
(DEPTNO CHAR(3) NOT NULL,
DEPTNAME VARCHAR(36) NOT NULL,
MGRNO CHAR(6) ,
ADMRDEPT CHAR(3) NOT NULL ,
LOCATION CHAR(16),
PRIMARY KEY (DEPTNO))
```

```
ALTER TABLE DEPARTMENT
ADD FOREIGN KEY ROD (ADMRDEPT)
REFERENCES DEPARTMENT
ON DELETE CASCADE
```

The following foreign key is added later

```
ALTER TABLE DEPARTMENT
ADD FOREIGN KEY RDE (MGRNO)
REFERENCES EMPLOYEE
ON DELETE SET NULL
```

The following indexes are created:

```
CREATE UNIQUE INDEX XDEPT1
ON DEPARTMENT (DEPTNO)
```

```
CREATE INDEX XDEPT2
ON DEPARTMENT (MGRNO)
```

```
CREATE INDEX XDEPT3
ON DEPARTMENT (ADMRDEPT)
```

The following alias is created for the table:

```
CREATE ALIAS DEPT FOR DEPARTMENT
```

The following table shows the content of the columns:

Table 41. Columns of the Department Table

Column Name	Description
DEPTNO	Department number or ID.
DEPTNAME	A name describing the general activities of the department.
MGRNO	Employee number (EMPNO) of the department manager.
ADMRDEPT	The department (DEPTNO) to which this department reports; the department at the highest level reports to itself.
LOCATION	Location of the department.

For a complete listing of DEPARTMENT, see “DEPARTMENT”.

DEPARTMENT

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	?

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
B01	PLANNING	000020	A00	?
C01	INFORMATION CENTER	000030	A00	?
D01	DEVELOPMENT CENTER	?	A00	?
D11	MANUFACTURING SYSTEMS	000060	D01	?
D21	ADMINISTRATION SYSTEMS	000070	D01	?
E01	SUPPORT SERVICES	000050	A00	?
E11	OPERATIONS	000090	E01	?
E21	SOFTWARE SUPPORT	000100	E01	?
F22	BRANCH OFFICE F2	?	E01	?
G22	BRANCH OFFICE G2	?	E01	?
H22	BRANCH OFFICE H2	?	E01	?
I22	BRANCH OFFICE I2	?	E01	?
J22	BRANCH OFFICE J2	?	E01	?

Employee Table (EMPLOYEE)

The employee table identifies all employees by an employee number and lists basic personnel information. The employee table is created with the following CREATE TABLE and ALTER TABLE statements:

```

CREATE TABLE EMPLOYEE
  (EMPNO      CHAR(6)          NOT NULL,
   FIRSTNME   VARCHAR(12)     NOT NULL,
   MIDINIT    CHAR(1)         NOT NULL,
   LASTNAME   VARCHAR(15)    NOT NULL,
   WORKDEPT   CHAR(3)         ,
   PHONENO    CHAR(4)         ,
   HIREDATE   DATE            ,
   JOB        CHAR(8)         ,
   EDLEVEL    SMALLINT       NOT NULL,
   SEX        CHAR(1)         ,
   BIRTHDATE  DATE            ,
   SALARY     DECIMAL(9,2)    ,
   BONUS      DECIMAL(9,2)    ,
   COMM       DECIMAL(9,2)    ,
   PRIMARY KEY (EMPNO))

ALTER TABLE EMPLOYEE
  ADD FOREIGN KEY RED (WORKDEPT)
  REFERENCES DEPARTMENT
  ON DELETE SET NULL

ALTER TABLE EMPLOYEE
  ADD CONSTRAINT NUMBER
  CHECK (PHONENO >= '0000' AND PHONENO <= '9999')
```

The following indexes are created:

```

CREATE UNIQUE INDEX XEMP1
  ON EMPLOYEE (EMPNO)

CREATE INDEX XEMP2
  ON EMPLOYEE (WORKDEPT)
```

The following alias is created for the table:

```
CREATE ALIAS EMP FOR EMPLOYEE
```

The table below shows the content of the columns.

Column Name	Description
EMPNO	Employee number
FIRSTNME	First name of employee
MIDINIT	Middle initial of employee
LASTNAME	Last name of employee
WORKDEPT	ID of department in which the employee works
PHONENO	Employee telephone number
HIREDATE	Date of hire
JOB	Job held by the employee
EDLEVEL	Number of years of formal education
SEX	Sex of the employee (M or F)
BIRTHDATE	Date of birth
SALARY	Yearly salary in dollars
BONUS	Yearly bonus in dollars
COMM	Yearly commission in dollars

For a complete listing of EMPLOYEE, see “EMPLOYEE” on page 357.

EMPLOYEE

EMP NO	FIRST NAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIRE DATE	JOB	ED LEVEL	SEX	BIRTH DATE	SAL-ARY	BONUS	COMM
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN		O'CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FILEREP	16	M	1932-08-11	19950	400	1596
000330	WING		LEE	E21	2103	1976-02-23	FILEREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FILEREP	16	M	1926-05-17	23840	500	1907
200010	DIAN	J	HEMMINGER	A00	3978	1965-01-01	SALESREP	18	F	1933-08-14	46500	1000	4220
200120	GREG		ORLANDO	A00	2167	1972-05-05	CLERK	14	M	1942-10-18	29250	600	2340
200140	KIM	N	NATZ	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
200170	KIYOSHI		YAMAMOTO	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
200220	REBA	K	JOHN	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
200240	ROBERT	M	MONTEVERDE	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
200280	EILEEN	R	SCHWARTZ	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
200310	MICHELLE	F	SPRINGER	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
200330	HELENA		WONG	E21	2103	1976-02-23	FIELDREP	14	F	1941-07-18	25370	500	2030
200340	ROY	R	ALONZO	E21	5698	1947-05-05	FIELDREP	16	M	1926-05-17	23840	500	1907

Employee Photo Table (EMP_PHOTO)

The employee photo table contains a photo for employees stored by employee number. The employee photo table is created with the following CREATE TABLE and ALTER TABLE statements:

```
CREATE TABLE EMP_PHOTO
  (EMPNO CHAR(6) NOT NULL,
   PHOTO_FORMAT VARCHAR(10) NOT NULL,
   PICTURE BLOB(100K),
   EMP_ROWID CHAR(40) NOT NULL DEFAULT '',
   PRIMARY KEY (EMPNO,PHOTO_FORMAT))
```

```
ALTER TABLE EMP_PHOTO
  ADD COLUMN DL_PICTURE DATALINK(1000)
  LINKTYPE URL NO LINK CONTROL
```

```
ALTER TABLE EMP_PHOTO
  ADD FOREIGN KEY (EMPNO)
  REFERENCES EMPLOYEE
  ON DELETE RESTRICT
```

The following index is created:

```
CREATE UNIQUE INDEX XEMP_PHOTO
ON EMP_PHOTO (EMPNO, PHOTO_FORMAT)
```

The table below shows the content of the columns.

Column Name	Description
EMPNO	Employee number
PHOTO_FORMAT	Format of image stored in PICTURE
PICTURE	Photo image
EMP_ROWID	Unique row id, not currently used

For a complete listing of EMP_PHOTO, see “EMP_PHOTO”.

EMP_PHOTO

EMPNO	PHOTO_FORMAT	PICTURE	EMP_ROWID
000130	bitmap	?	
000130	gif	?	
000140	bitmap	?	
000140	gif	?	
000150	bitmap	?	
000150	gif	?	
000190	bitmap	?	
000190	gif	?	

Employee ResumeTable (EMP_RESUME)

The employee photo table contains a resume for employees stored by employee number. The employee resumetable is created with the following CREATE TABLE and ALTER TABLE statements:

```
CREATE TABLE EMP_RESUME
(EMPNO CHAR(6) NOT NULL,
RESUME_FORMAT VARCHAR(10) NOT NULL,
RESUME CLOB(5K),
EMP_ROWID CHAR(40) NOT NULL DEFAULT '',
PRIMARY KEY (EMPNO, RESUME_FORMAT))
```

```
ALTER TABLE EMP_RESUME
ADD COLUMN DL_RESUME DATALINK(1000)
LINKTYPE URL NO LINK CONTROL
```

```
ALTER TABLE EMP_RESUME
ADD FOREIGN KEY (EMPNO)
REFERENCES EMPLOYEE
ON DELETE RESTRICT
```

The following index is created:

```
CREATE UNIQUE INDEX XEMP_RESUME
ON EMP_RESUME (EMPNO, RESUME_FORMAT)
```

The table below shows the content of the columns.

Column Name	Description
EMPNO	Employee number
RESUME_FORMAT	Format of text stored in RESUME
RESUME	Resume
EMP_ROWID	Unique row id, not currently used

For a complete listing of EMP_RESUME, see “EMP_RESUME”.

EMP_RESUME

EMPNO	RESUME_FORMAT	RESUME	EMP_ROWID
000130	ascii	?	
000130	html	?	
000140	ascii	?	
000140	html	?	
000150	ascii	?	
000150	html	?	
000190	ascii	?	
000190	html	?	

Employee to Project Activity Table (EMPPROJECT)

The employee to project activity table identifies the employee who performs each activity listed for each project. The employee’s level of involvement (full-time or part-time) and schedule for activity are also in the table. The employee to project activity table is created with the following CREATE TABLE and ALTER TABLE statements:

```

CREATE TABLE EMPPROJECT
  (EMPNO    CHAR(6)          NOT NULL,
   PROJNO   CHAR(6)          NOT NULL,
   ACTNO    SMALLINT         NOT NULL,
   EMPTIME  DECIMAL(5,2)    ,
   EMSTDATE DATE             ,
   EMENDATE DATE             )

ALTER TABLE EMPPROJECT
  ADD FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
  REFERENCES PROJACT
  ON DELETE RESTRICT

```

The following aliases are created for the table:

```

CREATE ALIAS EMPACT FOR EMPPROJECT

CREATE ALIAS EMP_ACT FOR EMPPROJECT

```

The table below shows the content of the columns.

Table 42. Columns of the Employee to Project Activity Table

Column Name	Description
EMPNO	Employee ID number

Table 42. Columns of the Employee to Project Activity Table (continued)

Column Name	Description
PROJNO	PROJNO of the project to which the employee is assigned
ACTNO	ID of an activity within a project to which an employee is assigned
EMPTIME	A proportion of the employee's full time (between 0.00 and 1.00) to be spent on the project from EMSTDATE to EMENDATE
EMSTDATE	Start date of the activity
EMENDATE	Completion date of the activity

For a complete listing of EMPPROJECT, see "EMPPROJECT".

EMPPROJECT

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000010	AD3100	10	.50	1982-01-01	1982-07-01
000070	AD3110	10	1.00	1982-01-01	1983-02-01
000230	AD3111	60	1.00	1982-01-01	1982-03-15
000230	AD3111	60	.50	1982-03-15	1982-04-15
000230	AD3111	70	.50	1982-03-15	1982-10-15
000230	AD3111	80	.50	1982-04-15	1982-10-15
000230	AD3111	180	.50	1982-10-15	1983-01-01
000240	AD3111	70	1.00	1982-02-15	1982-09-15
000240	AD3111	80	1.00	1982-09-15	1983-01-01
000250	AD3112	60	1.00	1982-01-01	1982-02-01
000250	AD3112	60	.50	1982-02-01	1982-03-15
000250	AD3112	60	1.00	1983-01-01	1983-02-01
000250	AD3112	70	.50	1982-02-01	1982-03-15
000250	AD3112	70	1.00	1982-03-15	1982-08-15
000250	AD3112	70	.25	1982-08-15	1982-10-15
000250	AD3112	80	.25	1982-08-15	1982-10-15
000250	AD3112	80	.50	1982-10-15	1982-12-01
000250	AD3112	180	.50	1982-08-15	1983-01-01
000260	AD3113	70	.50	1982-06-15	1982-07-01
000260	AD3113	70	1.00	1982-07-01	1983-02-01
000260	AD3113	80	1.00	1982-01-01	1982-03-01
000260	AD3113	80	.50	1982-03-01	1982-04-15
000260	AD3113	180	.50	1982-03-01	1982-04-15
000260	AD3113	180	1.00	1982-04-15	1982-06-01
000260	AD3113	180	1.00	1982-06-01	1982-07-01
000270	AD3113	60	.50	1982-03-01	1982-04-01
000270	AD3113	60	1.00	1982-04-01	1982-09-01
000270	AD3113	60	.25	1982-09-01	1982-10-15

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000270	AD3113	70	.75	1982-09-01	1982-10-15
000270	AD3113	70	1.00	1982-10-15	1983-02-01
000270	AD3113	80	1.00	1982-01-01	1982-03-01
000270	AD3113	80	.50	1982-03-01	1982-04-01
000030	IF1000	10	.50	1982-06-01	1983-01-01
000130	IF1000	90	1.00	1982-10-01	1983-01-01
000130	IF1000	100	.50	1982-10-01	1983-01-01
000140	IF1000	90	.50	1982-10-01	1983-01-01
000030	IF2000	10	.50	1982-01-01	1983-01-01
000140	IF2000	100	1.00	1982-01-01	1982-03-01
000140	IF2000	100	.50	1982-03-01	1982-07-01
000140	IF2000	110	.50	1982-03-01	1982-07-01
000140	IF2000	110	.50	1982-10-01	1983-01-01
000010	MA2100	10	.50	1982-01-01	1982-11-01
000110	MA2100	20	1.00	1982-01-01	1983-03-01
000010	MA2110	10	1.00	1982-01-01	1983-02-01
000200	MA2111	50	1.00	1982-01-01	1982-06-15
000200	MA2111	60	1.00	1982-06-15	1983-02-01
000220	MA2111	40	1.00	1982-01-01	1983-02-01
000150	MA2112	60	1.00	1982-01-01	1982-07-15
000150	MA2112	180	1.00	1982-07-15	1983-02-01
000170	MA2112	60	1.00	1982-01-01	1983-06-01
000170	MA2112	70	1.00	1982-06-01	1983-02-01
000190	MA2112	70	1.00	1982-01-01	1982-10-01
000190	MA2112	80	1.00	1982-10-01	1983-10-01
000160	MA2113	60	1.00	1982-07-15	1983-02-01
000170	MA2113	80	1.00	1982-01-01	1983-02-01
000180	MA2113	70	1.00	1982-04-01	1982-06-15
000210	MA2113	80	.50	1982-10-01	1983-02-01
000210	MA2113	180	.50	1982-10-01	1983-02-01
000050	OP1000	10	.25	1982-01-01	1983-02-01
000090	OP1010	10	1.00	1982-01-01	1983-02-01
000280	OP1010	130	1.00	1982-01-01	1983-02-01
000290	OP1010	130	1.00	1982-01-01	1983-02-01
000300	OP1010	130	1.00	1982-01-01	1983-02-01
000310	OP1010	130	1.00	1982-01-01	1983-02-01
000050	OP2010	10	.75	1982-01-01	1983-02-01
000100	OP2010	10	1.00	1982-01-01	1983-02-01
000320	OP2011	140	.75	1982-01-01	1983-02-01
000320	OP2011	150	.25	1982-01-01	1983-02-01
000330	OP2012	140	.25	1982-01-01	1983-02-01

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000330	OP2012	160	.75	1982-01-01	1983-02-01
000340	OP2013	140	.50	1982-01-01	1983-02-01
000340	OP2013	170	.50	1982-01-01	1983-02-01
000020	PL2100	30	1.00	1982-01-01	1982-09-15

Project Table (PROJECT)

The project table describes each project that the business is currently undertaking. Data contained in each row include the project number, name, person responsible, and schedule dates. The project table is created with the following CREATE TABLE and ALTER TABLE statements:

```
CREATE TABLE PROJECT
  (PROJNO CHAR(6) NOT NULL,
   PROJNAME VARCHAR(24) NOT NULL DEFAULT,
   DEPTNO CHAR(3) NOT NULL,
   RESPEMP CHAR(6) NOT NULL,
   PRSTAFF DECIMAL(5,2) ,
   PRSTDATE DATE ,
   PRENDATE DATE ,
   MAJPROJ CHAR(6) ,
   PRIMARY KEY (PROJNO))
```

```
ALTER TABLE PROJECT
  ADD FOREIGN KEY (DEPTNO)
  REFERENCES DEPARTMENT
  ON DELETE RESTRICT
```

```
ALTER TABLE PROJECT
  ADD FOREIGN KEY (RESPEMP)
  REFERENCES EMPLOYEE
  ON DELETE RESTRICT
```

```
ALTER TABLE PROJECT
  ADD FOREIGN KEY RPP (MAJPROJ)
  REFERENCES PROJECT
  ON DELETE CASCADE
```

The following indexes are created:

```
CREATE UNIQUE INDEX XPROJ1
  ON PROJECT (PROJNO)
```

```
CREATE INDEX XPROJ2
  ON PROJECT (RESPEMP)
```

The following alias is created for the table:

```
CREATE ALIAS PROJ FOR PROJECT
```

The table below shows the contents of the columns:

Column Name	Description
PROJNO	Project number
PROJNAME	Project name
DEPTNO	Department number of the department responsible for the project
RESPEMP	Employee number of the person responsible for the project

Column Name	Description
PRSTAFF	Estimated mean staffing
PRSTDATE	Estimated start date of the project
PRENDATE	Estimated end date of the project
MAJPROJ	Controlling project number for sub projects

For a complete listing of PROJECT, see "PROJECT".

PROJECT

PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
AD3100	ADMIN SERVICES	D01	000010	6.5	1982-01-01	1983-02-01	?
AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	1982-01-01	1983-02-01	AD3100
AD3111	PAYROLL PROGRAMMING	D21	000230	2	1982-01-01	1983-02-01	AD3110
AD3112	PERSONNEL PROGRAMMING	D21	000250	1	1982-01-01	1983-02-01	AD3110
AD3113	ACCOUNT PROGRAMMING	D21	000270	2	1982-01-01	1983-02-01	AD3110
IF1000	QUERY SERVICES	C01	000030	2	1982-01-01	1983-02-01	?
IF2000	USER EDUCATION	C01	000030	1	1982-01-01	1983-02-01	?
MA2100	WELD LINE AUTOMATION	D01	000010	12	1982-01-01	1983-02-01	?
MA2110	W L PROGRAMMING	D11	000060	9	1982-01-01	1983-02-01	MA2100
MA2111	W L PROGRAM DESIGN	D11	000220	2	1982-01-01	1982-12-01	MA2110
MA2112	W L ROBOT DESIGN	D11	000150	3	1982-01-01	1982-12-01	MA2110
MA2113	W L PROD CONT PROGS	D11	000160	3	1982-02-15	1982-12-01	MA2110
OP1000	OPERATION SUPPORT	E01	000050	6	1982-01-01	1983-02-01	?
OP1010	OPERATION	E11	000090	5	1982-01-01	1983-02-01	OP1000
OP2000	GEN SYSTEMS SERVICES	E01	000050	5	1982-01-01	1983-02-01	?
OP2010	SYSTEMS SUPPORT	E21	000100	4	1982-01-01	1983-02-01	OP2000
OP2011	SCP SYSTEMS SUPPORT	E21	000320	1	1982-01-01	1983-02-01	OP2010
OP2012	APPLICATIONS SUPPORT	E21	000330	1	1982-01-01	1983-02-01	OP2010
OP2013	DB/DC SUPPORT	E21	000340	1	1982-01-01	1983-02-01	OP2010
PL2100	WELD LINE PLANNING	B01	000020	1	1982-01-01	1982-09-15	MA2100

Project Activity Table (PROJECT)

The project activity table describes each project that the business is currently undertaking. Data contained in each row include the project number, activity number, and schedule dates. The project activity table is created with the following CREATE TABLE and ALTER TABLE statements:

```
CREATE TABLE PROJECT  
  (PROJNO CHAR(6) NOT NULL,  
   ACTNO SMALLINT NOT NULL,  
   ACSTAFF DECIMAL(5,2),  
   ACSTDATE DATE NOT NULL,  
   ACENDATE DATE ,  
   PRIMARY KEY (PROJNO, ACTNO, ACSTDATE))
```

```
ALTER TABLE PROJECT  
  ADD FOREIGN KEY RPAP (PROJNO)  
  REFERENCES PROJECT  
  ON DELETE RESTRICT
```

The following foreign key is added later:

```
ALTER TABLE PROJECT  
  ADD FOREIGN KEY RPAA (ACTNO)  
  REFERENCES ACT  
  ON DELETE RESTRICT
```

The following index is created:

```
CREATE UNIQUE INDEX XPROJAC1  
  ON PROJECT (PROJNO, ACTNO, ACSTDATE)
```

The table below shows the contents of the columns:

Column Name	Description
PROJNO	Project number
ACTNO	Activity number
ACSTAFF	Estimated mean staffing
ACSTDATE	Activity start date
ACENDATE	Activity end date

For a complete listing of PROJECT, see "PROJECT".

PROJECT

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3100	10	?	1982-01-01	?
AD3110	10	?	1982-01-01	?
AD3111	60	?	1982-01-01	?
AD3111	60	?	1982-03-15	?
AD3111	70	?	1982-03-15	?
AD3111	80	?	1982-04-15	?
AD3111	180	?	1982-10-15	?
AD3111	70	?	1982-02-15	?
AD3111	80	?	1982-09-15	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3112	60	?	1982-01-01	?
AD3112	60	?	1982-02-01	?
AD3112	60	?	1983-01-01	?
AD3112	70	?	1982-02-01	?
AD3112	70	?	1982-03-15	?
AD3112	70	?	1982-08-15	?
AD3112	80	?	1982-08-15	?
AD3112	80	?	1982-10-15	?
AD3112	180	?	1982-08-15	?
AD3113	70	?	1982-06-15	?
AD3113	70	?	1982-07-01	?
AD3113	80	?	1982-01-01	?
AD3113	80	?	1982-03-01	?
AD3113	180	?	1982-03-01	?
AD3113	180	?	1982-04-15	?
AD3113	180	?	1982-06-01	?
AD3113	60	?	1982-03-01	?
AD3113	60	?	1982-04-01	?
AD3113	60	?	1982-09-01	?
AD3113	70	?	1982-09-01	?
AD3113	70	?	1982-10-15	?
IF1000	10	?	1982-06-01	?
IF1000	90	?	1982-10-01	?
IF1000	100	?	1982-10-01	?
IF2000	10	?	1982-01-01	?
IF2000	100	?	1982-01-01	?
IF2000	100	?	1982-03-01	?
IF2000	110	?	1982-03-01	?
IF2000	110	?	1982-10-01	?
MA2100	10	?	1982-01-01	?
MA2100	20	?	1982-01-01	?
MA2110	10	?	1982-01-01	?
MA2111	50	?	1982-01-01	?
MA2111	60	?	1982-06-15	?
MA2111	40	?	1982-01-01	?
MA2112	60	?	1982-01-01	?
MA2112	180	?	1982-07-15	?
MA2112	70	?	1982-06-01	?
MA2112	70	?	1982-01-01	?
MA2112	80	?	1982-10-01	?
MA2113	60	?	1982-07-15	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
MA2113	80	?	1982-01-01	?
MA2113	70	?	1982-04-01	?
MA2113	80	?	1982-10-01	?
MA2113	180	?	1982-10-01	?
OP1000	10	?	1982-01-01	?
OP1010	10	?	1982-01-01	?
OP1010	130	?	1982-01-01	?
OP2010	10	?	1982-01-01	?
OP2011	140	?	1982-01-01	?
OP2011	150	?	1982-01-01	?
OP2012	140	?	1982-01-01	?
OP2012	160	?	1982-01-01	?
OP2013	140	?	1982-01-01	?
OP2013	170	?	1982-01-01	?
PL2100	30	?	1982-01-01	?

Activity Table (ACT)

The activity table describes each activity. The activity table is created with the following CREATE TABLE statement:

```
CREATE TABLE ACT
  (ACTNO SMALLINT NOT NULL,
   ACTKWD CHAR(6) NOT NULL,
   ACTDESC VARCHAR(20) NOT NULL,
   PRIMARY KEY (ACTNO))
```

The following indexes are created:

```
CREATE UNIQUE INDEX XACT1
  ON ACT (ACTNO)

CREATE UNIQUE INDEX XACT2
  ON ACT (ACTKWD)
```

The table below shows the contents of the columns.

Column Name	Description
ACTNO	Activity number
ACTKWD	Keyword for activity
ACTDESC	Description of activity

For a complete listing of ACT, see "ACT".

ACT

ACTNO	ACTKWD	ACTDESC
10	MANAGE	MANAGE/ADVISE
20	ECOST	ESTIMATE COST

30	DEFINE	DEFINE SPECS
40	LEADPR	LEAD PROGRAM/DESIGN
50	SPECS	WRITE SPECS
60	LOGIC	DESCRIBE LOGIC
70	CODE	CODE PROGRAMS
80	TEST	TEST PROGRAMS
90	ADMQS	ADM QUERY SYSTEM
100	TEACH	TEACH CLASSES
110	COURSE	DEVELOP COURSES
120	STAFF	PERS AND STAFFING
130	OPERAT	OPER COMPUTER SYS
140	MAINT	MAINT SOFTWARE SYS
150	ADMSYS	ADM OPERATING SYS
160	ADMDB	ADM DATA BASES
170	ADMDC	ADM DATA COMM
180	DOC	DOCUMENT

Class Schedule Table (CL_SCHED)

The class schedule table describes: each class, the start time for the class, the end time for the class, and the class code. The class schedule table is created with the following CREATE TABLE statement:

```
CREATE TABLE CL_SCHED
  (CLASS_CODE          CHAR(7),
   "DAY"              SMALLINT,
   STARTING           TIME,
   ENDING             TIME)
```

The table below gives the contents of the columns.

Column Name	Description
CLASS_CODE	Class code (room:teacher)
DAY	Day number of 4 day schedule
STARTING	Class start time
ENDING	Class end time

For a complete listing of CL_SCHED, see "CL_SCHED".

CL_SCHED

CLASS_CODE	DAY	STARTING	ENDING
042:BF	4	12:10:00	14:00:00
553:MJA	1	10:30:00	11:00:00
543:CWM	3	09:10:00	10:30:00
778:RES	2	12:10:00	14:00:00
044:HD	3	17:12:30	18:00:00

In Tray Table (IN_TRAY)

The in tray table describes an electronic in-basket containing: a timestamp from when the message was received, the user ID of the person sending the message, and the message itself. The in tray table is created with the following CREATE TABLE statement:

```
CREATE TABLE IN_TRAY
  (RECEIVED          TIMESTAMP,
   SOURCE            CHAR(8),
   SUBJECT           CHAR(64),
   NOTE_TEXT        VARCHAR(3000))
```

The table below gives the contents of the columns.

Column Name	Description
RECEIVED	Date and time received
SOURCE	User ID of person sending the note
SUBJECT	Brief description of the note
NOTE_TEXT	The note

For a complete listing of IN_TRAY, see "IN_TRAY".

IN_TRAY

RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
1988-12-25-17.12.30.000000	BADAMSON	FWD: Fantastic year! 4th Quarter Bonus.	To: JWALKER Cc: QUINTANA, NICHOLLS Jim, Looks like our hard work has paid off. I have some good beer in the fridge if you want to come over to celebrate a bit. Delores and Heather, are you interested as well? Bruce <Forwarding from ISTERN> Subject: FWD: Fantastic year! 4th Quarter Bonus. To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv <Forwarding from CHAAS> Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas

RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
1988-12-23-08.53.58.000000	ISTERN	FWD: Fantastic year! 4th Quarter Bonus.	To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv <Forwarding from CHAAS> Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas
1988-12-22-14.07.21.136421	CHAAS	Fantastic year! 4th Quarter Bonus.	To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas

Organization Table (ORG)

The organization table describes the organization of the corporation. The organization table is created with the following CREATE TABLE statement:

```
CREATE TABLE ORG
(DEPTNUMB SMALLINT NOT NULL,
DEPTNAME VARCHAR(14),
MANAGER SMALLINT,
DIVISION VARCHAR(10),
LOCATION VARCHAR(13))
```

The table below gives the contents of the columns.

Column Name	Description
DEPTNUMB	Department number
DEPTNAME	Department name
MANAGER	Manager number for the department
DIVISION	Division of the department
LOCATION	Location of the department

For a complete listing of ORG, see "ORG".

ORG

DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
10	Head Office	160	Corporate	New York
15	New England	50	Eastern	Boston

DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
20	Mid Atlantic	10	Eastern	Washington
38	South Atlantic	30	Eastern	Atlanta
42	Great Lakes	100	Midwest	Chicago
51	Plains	140	Midwest	Dallas
66	Pacific	270	Western	San Francisco
84	Mountain	290	Western	Denver

Staff Table (STAFF)

The staff table describes the employees. The staff table is created with the following CREATE TABLE statement:

```
CREATE TABLE STAFF
  (ID SMALLINT NOT NULL,
   NAME VARCHAR(9),
   DEPT SMALLINT,
   JOB CHAR(5),
   YEARS SMALLINT,
   SALARY DECIMAL(7,2),
   COMM DECIMAL(7,2))
```

The table below shows the contents of the columns.

Column Name	Description
ID	Employee number
NAME	Employee name
DEPT	Department number
JOB	Job title
YEARS	Years with the company
SALARY	Employee's annual salary
COMM	Employee's commision

For a complete listing of STAFF, see "STAFF".

STAFF

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	?
20	Pernal	20	Sales	8	18171.25	612.45
30	Marenghi	38	Mgr	5	17506.75	?
40	O'Brien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	?
60	Quigley	38	Sales	7	16508.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	?	13504.60	128.20
90	Koonitz	42	Sales	6	18001.75	1386.70
100	Plotz	42	Mgr	7	18352.80	?

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
110	Ngan	15	Clerk	5	12508.20	206.60
120	Naughton	38	Clerk	?	12954.75	180.00
130	Yamaguchi	42	Clerk	6	10505.90	75.60
140	Fraye	51	Mgr	6	21150.00	?
150	Williams	51	Sales	6	19456.50	637.65
160	Molinare	10	Mgr	7	22959.20	?
170	Kermisch	15	Clerk	4	12258.50	110.10
180	Abrahams	38	Clerk	3	12009.75	236.50
190	Sneider	20	Clerk	8	14252.75	126.50
200	Scoutten	42	Clerk	?	11508.60	84.20
210	Lu	10	Mgr	10	20010.00	?
220	Smith	51	Sales	7	17654.50	992.80
230	Lundquist	51	Clerk	3	13369.80	189.65
240	Daniels	10	Mgr	5	19260.25	?
250	Wheeler	51	Clerk	6	14460.00	513.30
260	Jones	10	Mgr	12	21234.00	?
270	Lea	66	Mgr	9	18555.50	?
280	Wilson	66	Sales	9	18674.50	811.50
290	Quill	84	Mgr	10	19818.00	?
300	Davis	84	Sales	5	15454.50	806.10
310	Graham	66	Sales	13	21000.00	200.30
320	Gonzales	66	Sales	4	16858.20	844.00
330	Burke	66	Clerk	1	10988.00	55.50
340	Edwards	84	Sales	7	17844.00	1285.00
3650	Gafney	84	Clerk	5	13030.50	188.00

Sales Table (SALES)

The sales table describes: each sales for each sales person. The sales table is created with the following CREATE TABLE statement:

```
CREATE TABLE SALES
(SALES_DATE DATE,
SALES_PERSON VARCHAR(15),
REGION VARCHAR(15),
SALES INTEGER)
```

The table below gives the contents of the columns.

Column Name	Description
SALES_DATE	Date the sale was made
SALES_PERSON	Person making the sale
REGION	Region where the sale was made
SALES	Number of sales

For a complete listing of SALES, see "SALES".

SALES

SALES_DATE	SALES_PERSON	REGION	SALES
12/31/1995	LUCCHESI	Ontario-South	1
12/31/1995	LEE	Ontario-South	3
12/31/1995	LEE	Quebec	1
12/31/1995	LEE	Manitoba	2
12/31/1995	GOUNOT	Quebec	1
03/29/1996	LUCCHESI	Ontario-South	3
03/29/1996	LUCCHESI	Quebec	1
03/29/1996	LEE	Ontario-South	2
03/29/1996	LEE	Ontario-North	2
03/29/1996	LEE	Quebec	3
03/29/1996	LEE	Manitoba	5
03/29/1996	GOUNOT	Ontario-South	3
03/29/1996	GOUNOT	Quebec	1
03/29/1996	GOUNOT	Manitoba	7
03/30/1996	LUCCHESI	Ontario-South	1
03/30/1996	LUCCHESI	Quebec	2
03/30/1996	LUCCHESI	Manitoba	1
03/30/1996	LEE	Ontario-South	7
03/30/1996	LEE	Ontario-North	3
03/30/1996	LEE	Quebec	7
03/30/1996	LEE	Manitoba	4
03/30/1996	GOUNOT	Ontario-South	2
03/30/1996	GOUNOT	Quebec	18
03/30/1996	GOUNOT	Manitoba	1
03/31/1996	LUCCHESI	Manitoba	1
03/31/1996	LEE	Ontario-South	14
03/31/1996	LEE	Ontario-North	3
03/31/1996	LEE	Quebec	7
03/31/1996	LEE	Manitoba	3
03/31/1996	GOUNOT	Ontario-South	2
03/31/1996	GOUNOT	Quebec	1
04/01/1996	LUCCHESI	Ontario-South	3
04/01/1996	LUCCHESI	Manitoba	1
04/01/1996	LEE	Ontario-South	8
04/01/1996	LEE	Ontario-North	?
04/01/1996	LEE	Quebec	8
04/01/1996	LEE	Manitoba	9
04/01/1996	GOUNOT	Ontario-South	3

SALES_DATE	SALES_PERSON	REGION	SALES
04/01/1996	GOUNOT	Ontario-North	1
04/01/1996	GOUNOT	Quebec	3
04/01/1996	GOUNOT	Manitoba	7

Appendix B. DB2 UDB for iSeries CL Command Descriptions

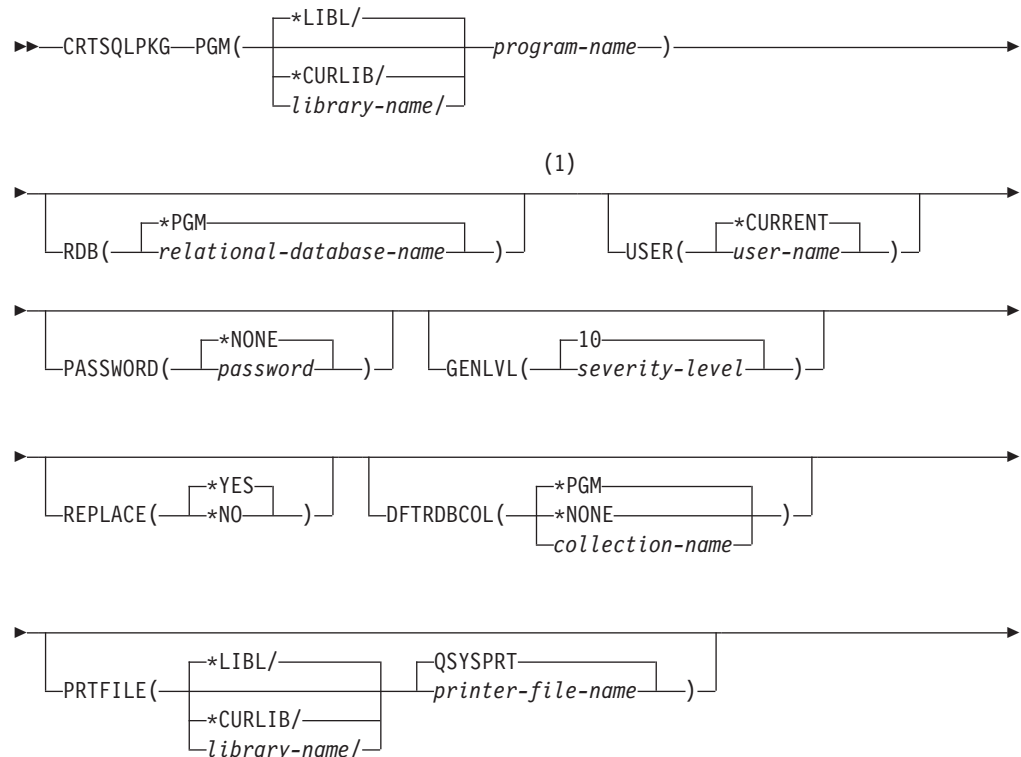
This appendix contains the syntax diagrams referred to and used in this book and the SQL Reference book.

For command descriptions, see the following topics:

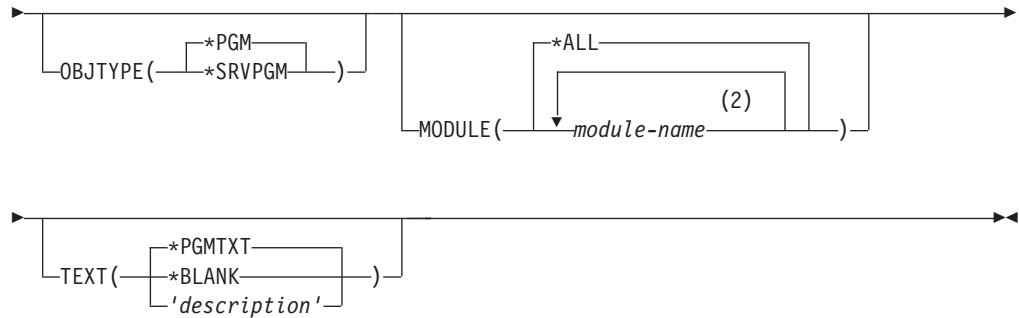
- “CRTSQLPKG (Create Structured Query Language Package) Command”
- “DLTSQLPKG (Delete Structured Query Language Package) Command” on page 379
- “PRTSQLINF (Print Structured Query Language Information) Command” on page 380
- “RUNSQLSTM (Run Structured Query Language Statement) Command” on page 381
- “STRSQL (Start Structured Query Language) Command” on page 391

CRTSQLPKG (Create Structured Query Language Package) Command

Job: B,I Pgm: B,I REXX: B,I Exec



CRTSQLPKG



Notes:

- 1 All parameters preceding this point can be specified in positional form.
- 2 A maximum of 256 modules may be specified.

Purpose:

The Create Structured Query Language Package (CRTSQLPKG) command is used to create (or re-create) an SQL package on a relational database from an existing distributed SQL program. A distributed SQL program is a program created by specifying the RDB parameter on a CRTSQLxxx (where xxx = C, CI, CBL, CBLI, FTN, PLI, or RPG or RPGI) command.

Parameters:

PGM

Specifies the qualified name of the program for which the SQL package is being created. The program must be a distributed SQL program.

The name of the program can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name: Specify the name of the library to be searched.

program-name: Specify the name of the program for which the package is being created.

RDB

Specifies the name of the relational database where the SQL package is being created.

***PGM:** The relational database name specified for the SQL program is used. The relational database name is specified on the RDB parameter of the distributed SQL program.

relational-database-name: Specify the name of the relational database where the SQL package is to be created. Use the Work with Relational Database Directory Entry (WRKRDBDIRE) command to show the relational database names that are valid on this parameter.

USER

Specifies the user name sent to the remote system when starting the conversation.

***CURRENT:** The user name associated with the current job is used.

user-name: Specify the user name being used for the application server job.

PASSWORD

Specifies the password to be used on the remote system.

***NONE:** No password is sent. If this value is specified, USER(*CURRENT) must also be specified.

password: Specify the password of the user name specified on the USER parameter.

GENLVL

Specifies the maximum severity level allowed for errors detected during SQL package creation. If errors occur at a level that exceeds the specified level, the SQL package is not created.

10: The default severity-level is 10.

severity-level: Specify the maximum severity level. Valid values range from 0 through 40.

REPLACE

Specifies whether an existing package is being replaced with the new package. more information about this parameter is in Appendix A, "Expanded Parameter Descriptions" in the *CL Reference* book.

***YES:** An existing SQL package of the same name is replaced by the new SQL package.

***NO:** An existing SQL package of the same name is not replaced; a new SQL package is not created if the package already exists in the specified library.

DFTRDBCOL

Specifies the collection name to be used for unqualified names of tables, views, indexes, and SQL packages. This parameter applies only to static SQL statements in the package.

***PGM:** The collection name specified for the SQL program is used. The default relational database collection name is specified on the DFTRDBCOL parameter of the distributed SQL program.

***NONE:** Unqualified names for tables, views, indexes, and SQL packages use the search conventions specified on the OPTION parameter of the CRTSQLxxx command used to create the program.

collection-name: Specify the collection name that is used for unqualified tables, views, indexes, and SQL packages.

PRTFILE

Specifies the qualified name of the printer device file to which the create SQL package error listing is directed. If no errors are detected during the creation of the SQL package, no listing is produced.

The name of the printer file can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name: Specify the name of the library to be searched.

CRTSQLPKG

QSYSPRT: If a file name is not specified, the create SQL package error listing is directed to the IBM-supplied printer file QSYSPRT.

printer-file-name: Specify the name of the printer device file to which the create SQL package error listing is directed.

OBJTYPE

Specifies the type of program for which an SQL package is created.

***PGM:** Create an SQL package from the program specified on the PGM parameter.

***SRVPGM:** Create an SQL package from the service program specified on the PGM parameter.

MODULE

Specifies a list of modules in a bound program.

***ALL:** An SQL package is created for each module in the program. An error message is sent if none of the modules in the program contain SQL statements or none of the modules is a distributed module.

Note: CRTSQLPKG can process programs that do not contain more than 1024 modules.

module-name: Specify the names of up to 256 modules in the program for which an SQL package is to be created. If more than 256 modules exist that need to have an SQL package created, multiple CRTSQLPKG commands must be used.

Duplicate module names in the same program are allowed. This command looks at each module in the program and if *ALL or the module name is specified on the MODULE parameter, processing continues to determine whether an SQL package should be created. If the module is created using SQL and the RDB parameter is specified on the precompile command, an SQL package is created for the module. The SQL package is associated with the module of the bound program.

TEXT

Specifies text that briefly describes the SQL package and its function.

***PGMTXT:** The text from the program for which the SQL package is being created is used.

***BLANK:** No text is specified.

'description': Specify a maximum of 50 characters of text, enclosed in apostrophes.

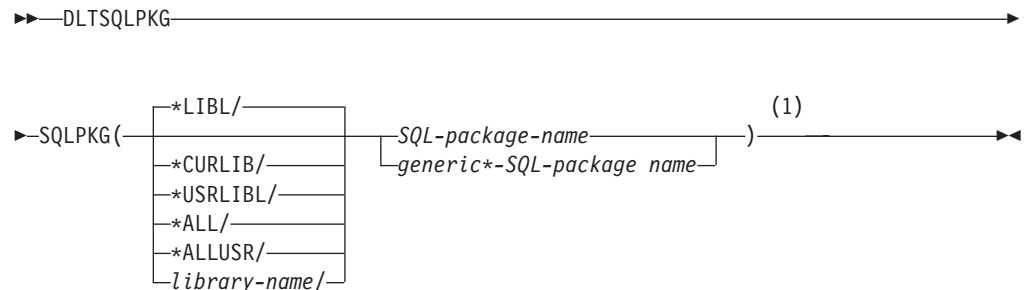
Example:

```
CRTSQLPKG PAYROLL RDB(SYSTEMA)
TEXT('Payroll Program')
```

This command creates an SQL package from the distributed SQL program PAYROLL on relational database SYSTEMA.

DLTSQLPKG (Delete Structured Query Language Package) Command

Job: B,I Pgm: B,I REXX: B,I Exec



Notes:

1 All parameters preceding this point can be specified in positional form.

Purpose:

The Delete Structured Query Language Package (DLTSQLPKG) command is used to delete one or more SQL packages.

DLTSQLPKG is a local command and must be used on the iSeries system where the SQL package being deleted is located.

To delete an SQL package on a remote system that is also an iSeries system, use the Submit Remote Command (SBMRMTCMD) command to run the DLTSQLPKG command on the remote system.

The user can do the following to delete an SQL package from a remote system that is not an iSeries system:

- Use interactive SQL to run the CONNECT and DROP PACKAGE operations.
- Sign on the remote system and use a command local to that system.
- Create and run an SQL program that contains a DROP PACKAGE SQL statement.

Parameters:

SQLPKG

Specifies the qualified name of the SQL package being deleted. A specific or generic SQL package name can be specified.

The name of the SQL Package can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

***USRLIBL:** Only the libraries in the user portion of the job's library list are searched.

***ALL:** All libraries in the system, including QSYS, are searched.

DLTSQLPKG

***ALLUSR:** All user libraries are searched. All libraries with names that do not begin with the letter Q are searched except for the following:

```
#CGULIB    #DFULIB    #RPGLIB    #SEULIB
#COBLIB    #DSULIB    #SDALIB
```

Although the following Qxxx libraries are provided by IBM, they typically contain user data that changes frequently. Therefore, these libraries are considered user libraries and are also searched:

```
QDSNX      QRCL      QUSRB RM   QUSRSYS
QG PL      QS36F    QUSRIJS   QUSRVxRxMx
QG PL38    QUSER38   QUSRINFSKR
QPFRDATA   QUSRADSM  QUSRRDARS
```

Note: A different library name, of the form QUSRVxRxMx, can be created by the user for each release that IBM supports. VxRxMx is the version, release, and modification level of the library.

library-name: Specify the name of the library to be searched.

SQL-package-name: Specify the name of the SQL package being deleted.

generic-SQL-package-name:* Specify the generic name of the SQL package to be deleted. A generic name is a character string of one or more characters followed by an asterisk (*); for example, ABC*. If a generic name is specified, all SQL packages with names that begin with the generic name, and for which the user has authority, are deleted. If an asterisk is not included with the generic (prefix) name, the system assumes it to be the complete SQL package name.

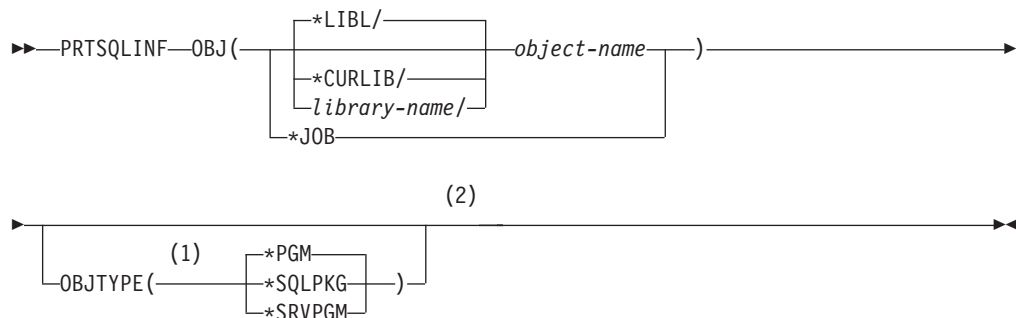
Example:

```
DLTSQLPKG SQLPKG(JONES)
```

This command deletes the SQL package JONES.

PRTSQLINF (Print Structured Query Language Information) Command

Job: B,I Pgm: B,I REXX: B,I Exec



Notes:

- 1 The OBJTYPE parameter is not allowed when OBJ(*JOB) is specified.
- 2 All parameters preceding this point can be specified in positional form.

Purpose:

The Print Structured Query Language Information (PRTSQLINF) command prints information about the SQL statements in a program, SQL packages, service program, or job. The information includes the SQL statements, the access plans used during the running of the statement, and a list of the command parameters which are defined either during the precompile of the source member for the object or when SQL statements are run.

Parameters:

OBJ

Specifies either the name of the object for which you want SQL information printed or '*JOB' indicating that the job's SQL information is to be printed. A named object can be a program, an SQL package, or a service program.

The name of the object can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name: Specify the name of the library to be searched.

object-name: Specify the name of the program, SQL package, or service program for which you want information printed.

- ***JOB:** Indicates that the SQL information for the current job is to be printed.

Optional parameters

OBJTYPE

Specifies the type of object.

- ***PGM:** The object is a program.
- ***SQLPKG:** The object is an SQL package.
- ***SRVPGM:** The object is a service program.

Example:

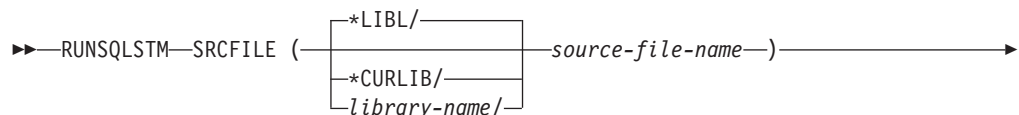
PRTSQLINF PAYROLL

This command prints information about the SQL statements contained in program PAYROLL.

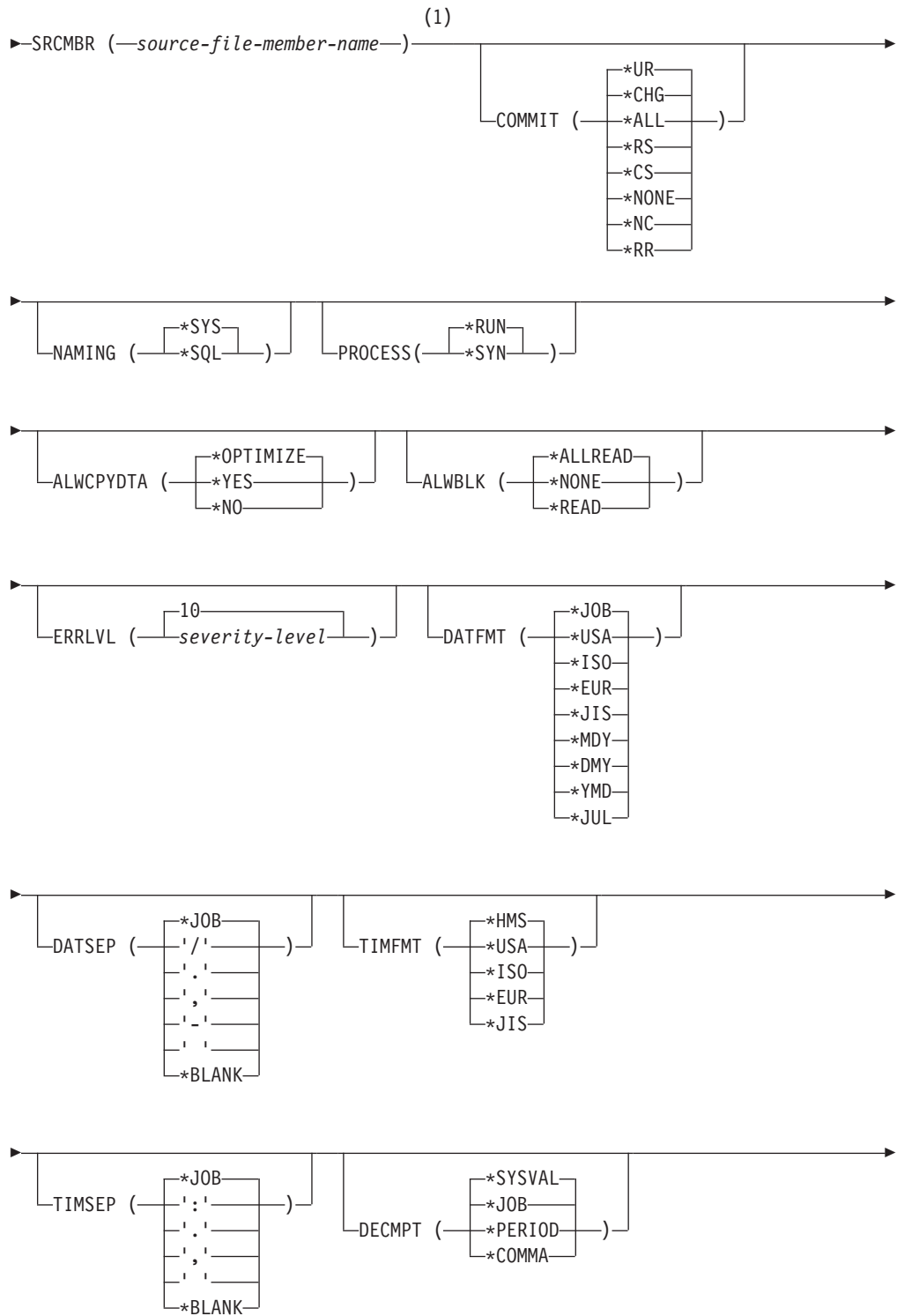
Note that OBJTYPE is now a dependent keyword and the prompt only occurs when the OBJ parameter is not *JOB.

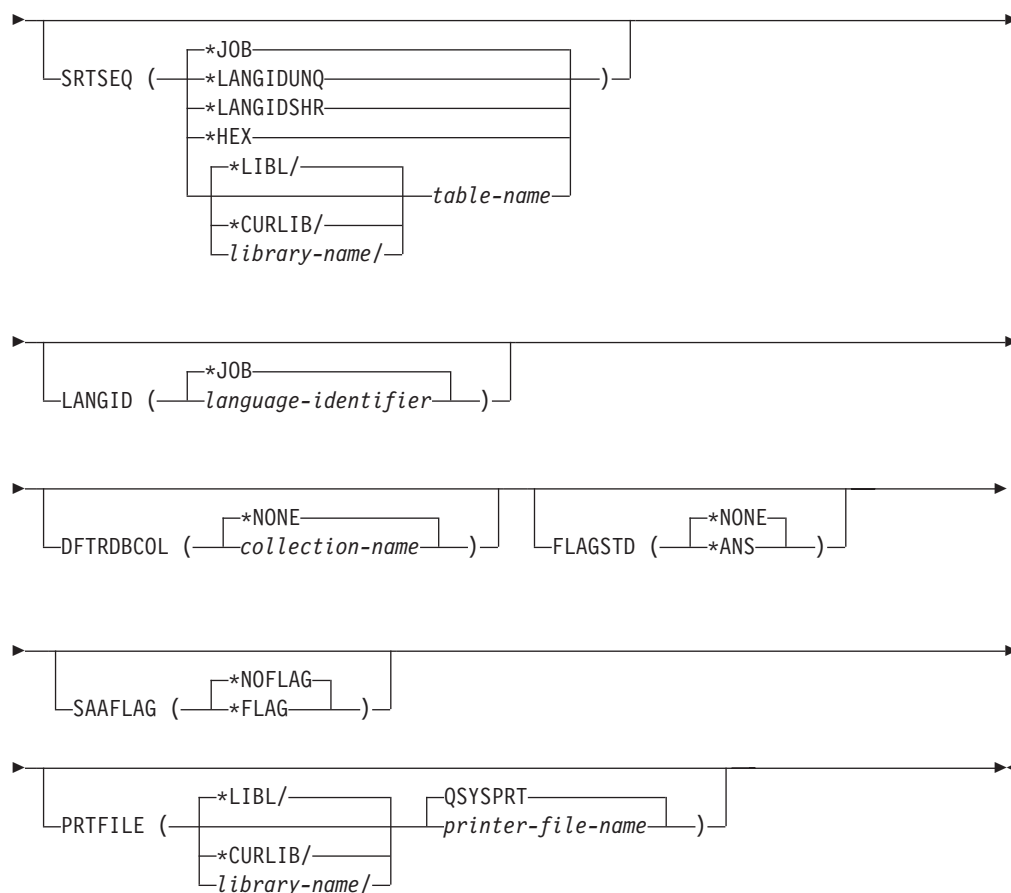
RUNSQLSTM (Run Structured Query Language Statement) Command

Job: B,I Pgm: B,I REXX: B,I Exec

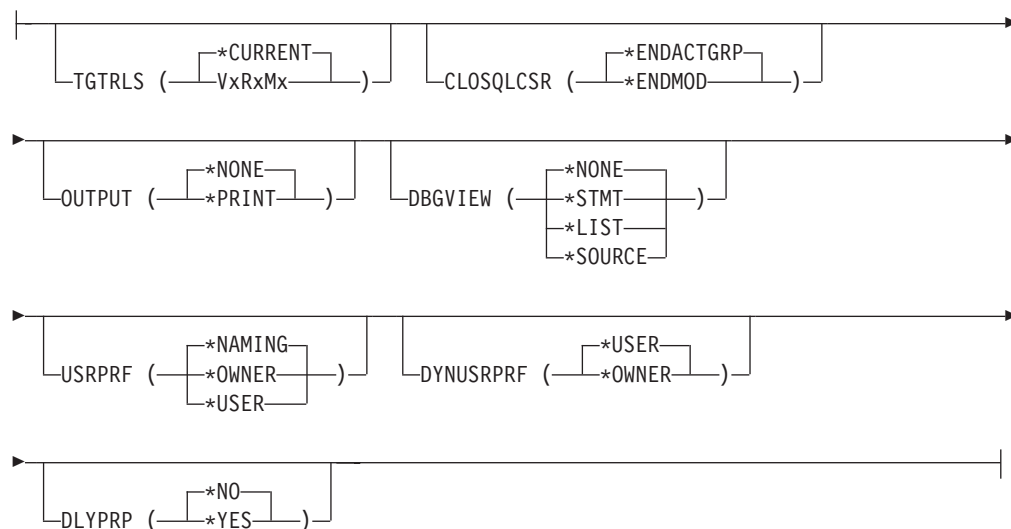


RUNSQLSTM





SQL-routine-parameters:



Notes:

- 1 All parameters preceding this point can be specified in positional form.

Purpose:

RUNSQLSTM

The Run Structured Query Language Statement (RUNSQLSTM) command processes a source file of SQL statements.

Parameters:

SRCFILE

Specifies the qualified name of the source file that contains the SQL statements to be run.

The name of the source file can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name: Specify the name of the library to be searched.

source-file-name: Specify the name of the source file that contains the SQL statements to be run. The source file can be a database file or an inline data file.

SRCMBR

Specifies the name of the source file member that contains the SQL statements to be run.

COMMIT

Specifies whether SQL statements in the source file are run under commitment control.

***CHG or *UR:** Specifies the objects referred to in SQL ALTER, CALL, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, RENAME, and REVOKE statements and the rows updated, deleted, and inserted are locked until the end of the unit of work (transaction). Uncommitted changes in other jobs can be seen.

***ALL or *RS:** Specifies the objects referred to in SQL ALTER, CALL, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, RENAME, and REVOKE statements and the rows selected, updated, deleted, and inserted are locked until the end of the unit of work (transaction). Uncommitted changes in other jobs cannot be seen.

***CS:** Specifies the objects referred to in SQL ALTER, CALL, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, RENAME, and REVOKE statements and the rows updated, deleted, and inserted are locked until the end of the unit of work (transaction). A row that is selected, but not updated, is locked until the next row is selected. Uncommitted changes in other jobs cannot be seen.

***NONE or *NC:** Specifies that commitment control is not used. Uncommitted changes in other jobs can be seen. If the SQL DROP COLLECTION statement is included in the program, *NONE or *NC must be used. If a relational database is specified on the RDB parameter and the relational database is on a system that is not on an AS/400, *NONE or *NC cannot be specified.

***RR:** Specifies the objects referred to in SQL ALTER, CALL, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, RENAME, and REVOKE statements and the rows selected, updated, deleted, and inserted are locked until the end of the unit of work (transaction). Uncommitted changes in other jobs cannot be seen. All tables referred to in SELECT, UPDATE, DELETE, and INSERT statements are locked exclusively until the end of the unit of work (transaction).

NAMING

Specifies the naming convention used for naming objects in SQL statements.

***SYS:** The system naming convention (library-name/file-name) is used.

***SQL:** The SQL naming convention (collection-name.table-name) is used.

PROCESS

Specifies whether SQL statements in the source file member are executed or syntax-checked only.

***RUN:** Statement are syntax-checked and run.

***SYN:** Statements are syntax-checked only.

ALWCPYDTA

Specifies whether a copy of the data can be used in a SELECT statement.

***OPTIMIZE:** The system determines whether to use the data retrieved directly from the database or to use a copy of the data. The decision is based on which method provides the best performance. If COMMIT is *CHG or *CS and ALWBLK is not *ALLREAD, or if COMMIT is *ALL or *RR, then a copy of the data is used only when it is necessary to run a query.

***YES:** A copy of the data is used only when necessary.

***NO:** A copy of the data is not used. If temporary copy of the data is required to perform the query, an error message is returned.

ALWBLK

Specifies whether the database manager can use record blocking, and the extent to which blocking can be used for read-only cursors.

***ALLREAD:** Rows are blocked for read-only cursors if *NONE or *CHG is specified on the COMMIT parameter. All cursors in a program that are not explicitly able to be updated are opened for read-only processing even though EXECUTE or EXECUTE IMMEDIATE statements may be in the program.

Specifying *ALLREAD:

- Allows record blocking under commitment control level *CHG in addition to the blocking allowed for *READ.
- Can improve the performance of almost all read-only cursors in programs, but limits queries in the following ways:
 - The Rollback (ROLLBACK) command, a ROLLBACK statement in host languages, or the ROLLBACK HOLD SQL statement does not reposition a read-only cursor when *ALLREAD is specified.
 - Dynamic running of a positioned UPDATE or DELETE statement (for example, using EXECUTE IMMEDIATE), cannot be used to update a row in a cursor unless the DECLARE statement for the cursor includes the FOR UPDATE clause.

***NONE:** Rows are not blocked for retrieval of data for cursors.

Specifying *NONE:

- Guarantees that the data retrieved is current.
- May reduce the amount of time required to retrieve the first row of data for a query.
- Stops the database manager from retrieving a block of data rows that is not used by the program when only the first few rows of a query are retrieved before the query is closed.

RUNSQLSTM

- Can degrade the overall performance of a query that retrieves a large number of rows.
- ***READ:** Records are blocked for read-only retrieval of data for cursors when:
 - *NONE is specified on the COMMIT parameter, which indicates that commitment control is not used.
 - The cursor is declared with a FOR FETCH ONLY clause or there are no dynamic statements that could run a positioned UPDATE or DELETE statement for the cursor.

Specifying *READ can improve the overall performance of queries that meet the above conditions and retrieve a large number of records.

ERRLVL

Specifies whether the processing is successful, based on the severity of the messages generated by the processing of the SQL statements. If errors that are greater than the value specified on this parameter occur during processing, no more statements are processed and the statements are rolled back if they are running under commitment control.

10: Statement processing is stopped when error messages with a severity level greater than 10 are received.

severity-level: Specify the severity level to be used.

DATFMT

Specifies the format used when accessing date result columns. For input date strings, the specified value is used to determine whether the date is specified in a valid format.

Note: An input date string that uses the format *USA, *ISO, *EUR, or *JIS is always valid.

***JOB:** The format specified for the job is used. Use the Display Job (DSPJOB) command to determine the current date format for the job.

***USA:** The United States date format (mm/dd/yyyy) is used.

***ISO:** The International Organization for Standardization (ISO) date format (yyyy-mm-dd) is used.

***EUR:** The European date format (dd.mm.yyyy) is used.

***JIS:** The Japanese Industrial Standard date format (yyyy-mm-dd) is used.

***MDY:** The date format (mm/dd/yy) is used.

***DMY:** The date format (dd/mm/yy) is used.

***YMD:** The date format (yy/mm/dd) is used.

***JUL:** The Julian date format (yy/ddd) is used.

DATSEP

Specifies the separator used when accessing date result columns.

Note: This parameter applies only when *JOB, *MDY, *DMY, *YMD, or *JUL is specified on the DATFMT parameter.

***JOB:** The date separator specified for the job is used. Use the Display Job (DSPJOB) command to determine the current value for the job.

'/': A slash (/) is used.

':': A period (.) is used.

',': A comma (,) is used.

'-': A dash (-) is used.

' ': A blank () is used.

***BLANK:** A blank () is used.

TIMFMT

Specifies the format used when accessing time result columns. For input time strings, the specified value is used to determine whether the time is specified in a valid format.

Note: An input date string that uses the format *USA, *ISO, *EUR, or *JIS is always valid.

***HMS:** The hh:mm:ss format is used.

***USA:** The United States time format hh:mm xx is used, where xx is AM or PM.

***ISO:** The International Organization for Standardization (ISO) time format hh.mm.ss is used.

***EUR:** The European time format hh.mm.ss is used.

***JIS:** The Japanese Industrial Standard time format hh:mm:ss is used.

TIMSEP

Specifies the separator used when accessing time result columns.

Note: This parameter applies only when *HMS is specified on the TIMFMT parameter.

***JOB:** The time separator specified for the job is used. Use the Display Job (DSPJOB) command to determine the current value for the job.

' ': A colon (:) is used.

':': A period (.) is used.

',': A comma (,) is used.

' ': A blank () is used.

***BLANK:** A blank () is used.

DECMPT

Specifies the decimal point value used for numeric constants in SQL statements.

RUNSQLSTM

***JOB:** The value used as the decimal point for numeric constants in SQL is the representation of decimal point specified by the job running the statement.

***SYSVAL:** The QDECFMT system value is used as the decimal point.

***PERIOD:** A period represents the decimal point.

***COMMA:** A comma represents the decimal point.

SRTSEQ

Specifies the sort sequence table to be used for string comparisons in SQL statements.

***JOB:** The LANGID value for the job is retrieved.

***LANGIDSHR:** The sort sequence table uses the same weight for multiple characters, and is the shared-weight sort sequence table associated with the language specified on the LANGID parameter.

***LANGIDUNQ:** The unique-weight sort table for the language specified on the LANGID parameter is used.

***HEX:** A sort sequence table is not used. The hexadecimal values of the characters are used to determine the sort sequence.

The name of the table name can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name: Specify the name of the library to be searched.

table-name: Specify the name of the sort sequence table to be used.

LANGID

Specifies the language identifier to be used when SRTSEQ(*LANGIDUNQ) or SRTSEQ(*LANGIDSHR) is specified.

***JOB:** The LANGID value for the job is retrieved during the precompile.

language-identifier: Specify a language identifier.

DFTRDBCOL

Specifies the collection name used for the unqualified names of tables, views, indexes, and SQL packages.

***NONE:** The naming convention defined on the OPTION parameter is used.

collection-name: Specify the name of the collection identifier. This value is used instead of the naming convention specified on the OPTION parameter.

FLAGSTD

Specifies the American National Standards Institute (ANSI) flagging function. This parameter flags SQL statements to verify whether they conform to the following standards.

ANSI X3.135-1992 entry

ISO 9075-1992 entry

FIPS 127.2 entry

***NONE:** The SQL statements are not checked to determine whether they conform to ANSI standards.

***ANS:** The SQL statements are checked to determine whether they conform to ANSI standards.

SAAFLAG

Specifies the IBM SQL flagging function. This parameter flags SQL statements to verify whether they conform to IBM SQL syntax. More information about which IBM database products IBM SQL syntax is in the *DRDA IBM SQL Reference*, SC26-3255-00.

***NOFLAG:** The SQL statements are not checked to determine whether they conform to IBM SQL syntax.

***FLAG:** The SQL statements are checked to determine whether they conform to IBM SQL syntax.

PRTFILE

Specifies the qualified name of the printer device file to which the RUNSQLSTM printout is directed. The file must have a minimum length of 132 bytes. If a file with a record length of less than 132 bytes is specified, information is lost.

The name of the printer file can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name: Specify the name of the library to be searched.

QSYSPRT: If a file name is not specified, the RUNSQLSTM printout is directed to the IBM-supplied printer file QSYSPRT.

printer-file-name: Specify the name of the printer device file to which the RUNSQLSTM printout is directed.

Parameters for SQL routines:

The parameters listed below only apply to statements within the source file that create SQL procedures, SQL functions, and SQL triggers. The parameters are used during the creation of the program object associated with SQL procedures, SQL functions, and SQL triggers.

TGTRLS

Specifies the release of the operating system on which the user intends to use the object being created.

In the examples given for the *CURRENT value, and when specifying the *release-level* value, the format VxRxMx is used to specify the release, where Vx is the version, Rx is the release, and Mx is the modification level. For example, V2R3M0 is version 2, release 3, modification level 0.

***CURRENT** The object is to be used on the release of the operating system currently running on the user's system. For example, if V2R3M5 is running on the system, *CURRENT means the user intends to use the object on a system with V2R3M5 installed. The user can also use the object on a system with any subsequent release of the operating system installed.

RUNSQLSTM

Note: If V2R3M5 is running on the system, and the object is to be used on a system with V2R3M0 installed, specify TGTRLS(V2R3M0) not TGRRLS(*CURRENT).

release-level: Specify the release in the format VxRxMx. The object can be used on a system with the specified release or with any subsequent release of the operating system installed.

Valid values depend on the current version, release, and modification level, and they change with each new release. If you specify a release-level which is earlier than the earliest release level supported by this command, an error message is sent indicating the earliest supported release.

CLOSQLCSR

Specifies when SQL cursors are implicitly closed, SQL prepared statements are implicitly discarded, and LOCK TABLE locks are released. SQL cursors are explicitly closed when you issue the CLOSE, COMMIT, or ROLLBACK (without HOLD) SQL statements.

***ENDACTGRP:** SQL cursors are closed and SQL prepared statements are implicitly discarded.

ENDMOD: SQL cursors are closed and SQL prepared statements are implicitly discarded when the module is exited. LOCK TABLE locks are released when the first SQL program on the call stack ends.

OUTPUT

Specifies whether the precompiler listing is generated.

***NONE:** The precompiler listing is not generated.

***PRINT:** The precompiler listing is generated.

DBGVIEW

Specifies the type of source debug information to be provided by the SQL precompiler.

***NONE:** The source view will not be generated.

***STMT:** Allows the compiled module to be debugged using program statement numbers and symbolic identifiers.

***LIST:** Generates the listing view for debugging the compiled module object.

***SOURCE:** The source view for SQL procedures, functions, and triggers will be generated.

USRPRF

Specifies the user profile that is used when the compiled program object is run, including the authority that the program object has for each object in static SQL statements. The profile of either the program owner or the program user is used to control which objects can be used by the program object.

***NAMING:** The user profile is determined by the naming convention. If the naming convention is *SQL, USRPRF(*OWNER) is used. If the naming convention is *SYS, USRPRF(*USER) is used.

***USER:** The profile of the user running the program object is used.

***OWNER:** The user profiles of both the program owner and the program user are used when the program is run.

DYNUSRPRF

Specifies the user profile to be used for dynamic SQL statements.

***USER:** For local, dynamic SQL statements run under the user of the program's user. For distributed, dynamic SQL statements run under the profile of the SQL package's user.

***OWNER:** For local, dynamic SQL statements run under the profile of the program's owner. For distributed, dynamic SQL statements run under the profile of the SQL package's owner.

DLYPRP

Specifies whether the dynamic statement validation for a PREPARE statement is delayed until an OPEN, EXECUTE, or DESCRIBE statement is run. Delaying validation improves performance by eliminating redundant validation.

***NO:** Dynamic statement validation is not delayed. When the dynamic statement is prepared, the access plan is validated. When the dynamic statement is used in an OPEN or EXECUTE statement, the access plan is revalidated. Because the authority or the existence of objects referred to by the dynamic statement may change, you must still check the SQLCODE or SQLSTATE after issuing the OPEN or EXECUTE statement to ensure that the dynamic statement is still valid.

***YES:** Dynamic statement validation is delayed until the dynamic statement is used in an OPEN, EXECUTE, or DESCRIBE SQL statement. When the dynamic statement is used, the validation is completed and an access plan is built. If you specify *YES on this parameter, you should check the SQLCODE and SQLSTATE after running an OPEN, EXECUTE, or DESCRIBE statement to ensure that the dynamic statement is valid.

Note: If you specify *YES, performance is not improved if the INTO clause is used on the PREPARE statement or if a DESCRIBE statement uses the dynamic statement before an OPEN is issued for the statement.

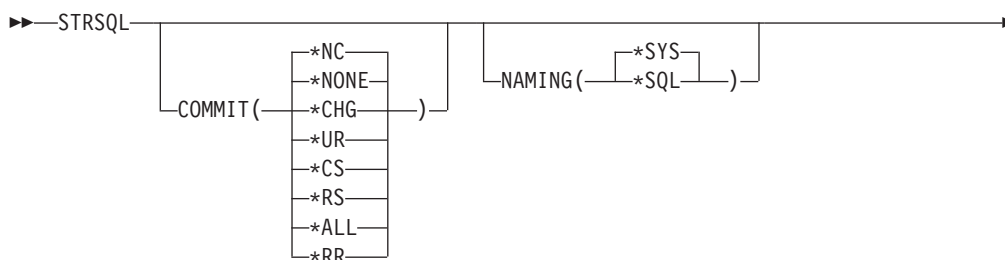
Example:

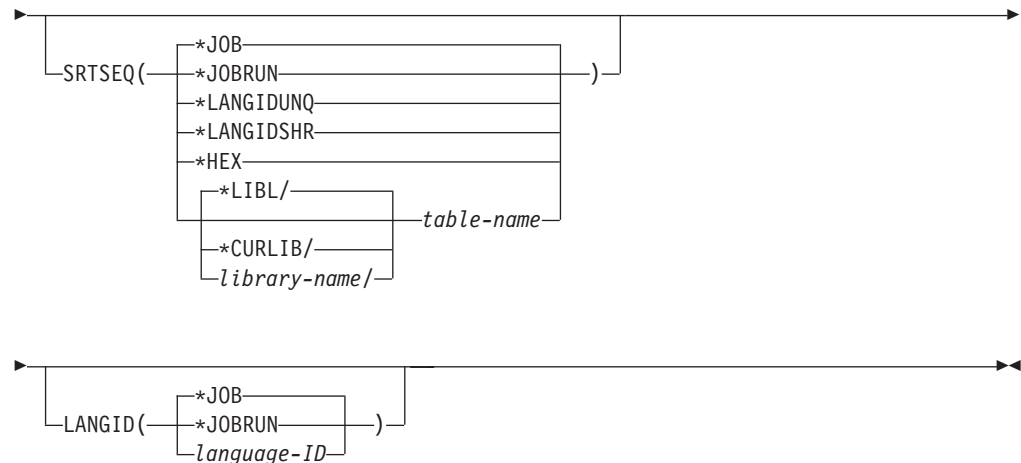
```
RUNSQLSTM SRCFILE(MYLIB/MYFILE) SRCMBR(MYMBR)
```

This command processes the SQL statements in member MYMBR found in file MYFILE in library MYLIB.

STRSQL (Start Structured Query Language) Command

Job: I Pgm: I REXX: I Exec



**Notes:**

- 1 All parameters preceding this point can be specified in positional form.
- 2 DATSEP is only valid when *MDY, *DMY, *YMD, or *JUL is specified on the DATFMT parameter.
- 3 TIMSEP is only valid when TIMFMT(*HMS) is specified.
- 4 PGMLNG and SQLSTRDLM are valid only when PROCESS(*SYN) is specified.
- 5 PGMLNG and SQLSTRDLM are valid only when PROCESS(*SYN) is specified.
- 6 SQLSTRDLM is valid only when PGMLNG(*CBL) is specified.

Purpose:

The Start Structured Query Language (STRSQL) command starts the interactive Structured Query Language (SQL) program. The program starts the statement entry of the interactive SQL program which immediately shows the Enter SQL Statements display. This display allows the user to build, edit, enter, and run an SQL statement in an interactive environment. Messages received during the running of the program are shown on this display.

Parameters:**COMMIT**

Specifies whether the SQL statements are run under commitment control.

***NONE or *NC:** Specifies that commitment control is not used. Uncommitted changes in other jobs can be seen. If the SQL DROP COLLECTION statement is included in the program, *NONE or *NC must be used. If a relational database is specified on the RDB parameter and the relational database is on a system that is not on an iSeries, *NONE or *NC cannot be specified.

***CHG or *UR:** Specifies the objects referred to in SQL ALTER, CALL, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, RENAME, and REVOKE statements and the rows updated, deleted, and inserted are locked until the end of the unit of work (transaction). Uncommitted changes in other jobs can be seen.

STRSQL

***CS:** Specifies the objects referred to in SQL ALTER, CALL, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, RENAME, and REVOKE statements and the rows updated, deleted, and inserted are locked until the end of the unit of work (transaction). A row that is selected, but not updated, is locked until the next row is selected. Uncommitted changes in other jobs cannot be seen.

***ALL or *RS:** Specifies the objects referred to in SQL ALTER, CALL, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, RENAME, and REVOKE statements and the rows selected, updated, deleted, and inserted are locked until the end of the unit of work (transaction). Uncommitted changes in other jobs cannot be seen.

***RR:** Specifies the objects referred to in SQL ALTER, CALL, COMMENT ON, CREATE, DROP, GRANT, LABEL ON, RENAME, and REVOKE statements and the rows selected, updated, deleted, and inserted are locked until the end of the unit of work (transaction). Uncommitted changes in other jobs cannot be seen. All tables referred to in SELECT, UPDATE, DELETE, and INSERT statements are locked exclusively until the end of the unit of work (transaction).

Note: The default for this parameter for the CRTSQLXXX commands (when XXX=CI, CPPI, CBL, FTN, PLI, CBLI, RPG or RPGI) is *CHG.

NAMING

Specifies the naming convention used for naming objects in SQL statements.

***SYS:** The system naming convention (library-name/file-name) is used.

***SQL:** The SQL naming convention (collection-name.table-name) is used.

PROCESS

Specifies the values used to process the SQL statements.

***RUN:** The statements are syntax checked, data checked, and then run.

***VLD:** The statements are syntax checked and data checked, but not run.

***SYN:** The statements are syntax checked only.

LIBOPT

Specifies which collections and libraries are used as a basis for building a collection list when the F4, F16, F17, or F18 function key is pressed.

The name of the collection list can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

***USRLIBL:** Only the libraries in the user portion of the job's library list are searched.

***ALL:** All libraries in the system, including QSYS, are searched.

***ALLUSR:** All user libraries are searched. All libraries with names that do not begin with the letter Q are searched except for the following:

#CGULIB	#DFULIB	#RPLIB	#SEULIB
#COBLIB	#DSULIB	#SDALIB	

Although the following Qxxx libraries are provided by IBM, they typically contain user data that changes frequently. Therefore, these libraries are considered user libraries and are also searched:

QDSNX	QRCL	QUSRBRM	QUSRSYS
QGPL	QS36F	QUSRIJS	QUSRVxRxMx
QGPL38	QUSER38	QUSRINFSKR	
QPFRDATA	QUSRADSM	QUSRRDARS	

Note: A different library name, of the form QUSRVxRxMx, can be created by the user for each release that IBM supports. VxRxMx is the version, release, and modification level of the library.

library-name: Specify the name of the library to be searched.

LISTTYPE

Specifies the types of objects that are displayed with list support by pressing the F4, F16, F17, or F18 function key.

***ALL:** All objects are displayed.

***SQL:** Only SQL-created objects are displayed.

REFRESH

Specifies when the display select output data is refreshed.

***ALWAYS:** Data is normally refreshed during forward and backward scrolling.

***FORWARD:** Data is refreshed only during forward scrolling to the end of the data for the first time. When scrolling backward, a copy of the data already viewed is shown.

ALWCPYDTA

Specifies whether a copy of the data can be used in a SELECT statement. If COMMIT(*ALL) is specified, SQL run time ignores the ALWCPYDTA value and uses current data.

***YES:** A copy of the data is used when necessary.

***OPTIMIZE:** The system determines whether to use the data retrieved from the database or to use a copy of the data. The determination is based on which will provide the best performance.

***NO:** A copy of the data is not allowed. If a temporary copy of the data is required to perform the query, an error message is returned.

DATFMT

Specifies the date format used in SQL statements.

***JOB:** The format specified on the job attribute DATFMT is used.

***USA:** The United States date format (mm/dd/yyyy) is used.

***ISO:** The International Standards Organization date format (yyyy-mm-dd) is used.

***EUR:** The European date format (dd.mm.yyyy) is used.

***JIS:** The Japanese Industry Standard Christian Era date format (yyyy-mm-dd) is used.

***MDY:** The month, day, and year date format (mm/dd/yy) is used.

***DMY:** The day, month, and year date format (dd/mm/yy) is used.

***YMD:** The year, month, and day date format (yy/mm/dd) is used.

***JUL:** The Julian date format (yy/ddd) is used.

STRSQL

DATSEP

Specifies the date separator used in SQL statements.

***JOB:** The date separator specified on the job attribute is used. If the user specifies *JOB on a new interactive SQL session, the current value is stored and used. Later changes to the job's date separator are not detected by interactive SQL.

***BLANK:** A blank () is used.

'/: A slash (/) is used.

':: A period (.) is used.

',': A comma (,) is used.

'-: A dash (-) is used.

' ':: A blank () is used.

TIMFMT

Specifies the time format used in SQL statements.

***HMS:** The Hour-Minute-Second time format (hh:mm:ss) is used.

***USA:** The United States time format (hh:mm xx, where xx is AM or PM) is used.

***ISO:** The International Standards Organization time format (hh.mm.ss) is used.

***EUR:** The European time format (hh.mm.ss) is used.

***JIS:** The Japanese Industry Standard Christian Era time format (hh:mm:ss) is used.

TIMSEP

Specifies the time separator used in SQL statements.

***JOB:** The time separator specified on the job attribute is used. If the user specifies *JOB on a new interactive SQL session, the current value is stored and used. Later changes to the job's time separator are not detected by interactive SQL.

***BLANK:** A blank () is used.

':: A colon (:) is used.

':: A period (.) is used.

',': A comma (,) is used.

' ':: A blank () is used.

DECPNT

Specifies the kind of decimal point to use.

***JOB:** The value used as the decimal point for numeric constants in SQL is the representation of decimal point specified for the job running the statement.

***SYSVAL:** The decimal point is extracted from the system value. If the user specifies *SYSVAL on a new interactive SQL session, the current value is stored and used. Later changes to the system's time separator are not detected by interactive SQL.

***PERIOD:** A period represents the decimal point.

***COMMA:** A comma represents the decimal point.

PGMLNG

Specifies which program language syntax rules to use. To use this parameter, *SYN must be selected at the PROCESS parameter.

***NONE:** No specific language's syntax check rules are used.

The supported languages are:

***C:** Syntax checking is done according to the C language syntax rules.

***CBL:** Syntax checking is done according to the COBOL language syntax rules.

***PLI:** Syntax checking is done according to the PL/I language syntax rules.

***RPG:** Syntax checking is done according to the RPG language syntax rules.

***FTN:** Syntax checking is done according to the FORTRAN language syntax rules.

SQLSTRDLM

Specifies the SQL string delimiter. Use of this parameter requires using the COBOL (*CBL) character set.

***QUOTESQL:** A quotation mark represents the SQL string delimiter.

***APOSTSQL:** An apostrophe represents the SQL string delimiter.

SRTSEQ

Specifies the sort sequence table to be used for string comparisons in SQL statements on the Enter SQL Statements display.

***JOB:** The SRTSEQ value for the job is retrieved.

***JOBRUN:** The SRTSEQ value for the job is retrieved each time the user starts interactive SQL.

***LANGIDUNQ:** The unique-weight sort table for the language specified on the LANGID parameter is used.

***LANGIDSHR:** The shared-weight sort table for the language specified on the LANGID parameter is used.

***HEX:** A sort sequence table is not used. The hexadecimal values of the characters are used to determine the sort sequence.

The name of the table name can be qualified by one of the following library values:

***LIBL:** All libraries in the job's library list are searched until the first match is found.

***CURLIB:** The current library for the job is searched. If no library is specified as the current library for the job, the QGPL library is used.

library-name: Specify the name of the library to be searched.

table-name: Specify the name of the sort sequence table to be used with the interactive SQL session.

LANGID

Specifies the language identifier to be used when SRTSEQ(*LANGIDUNQ) or SRTSEQ(*LANGIDSHR) is specified.

***JOB:** The LANGID value for the job is retrieved.

***JOBRUN:** The LANGID value for the job is retrieved each time interactive SQL is started.

language-ID: Specify the language identifier to be used.

STRSQL


Example:


```
STRSQL PROCESS(*SYN) NAMING(*SQL)
        DECPNT(*COMMA) PGMLNG(*CBL)
        SQLSTRDLM(*APOSTSQL)
```


This command starts an interactive SQL session that checks only the syntax of SQL statements. The character set used by the syntax checker uses the COBOL language syntax rules. The SQL naming convention is used for this session. The decimal point is represented by a comma, and the SQL string delimiter is represented by an apostrophe.


Bibliography


This guide lists publications that provide additional information about topics described or referred to in this guide. The manuals in this section are listed with their full title and order number, but when referred to in text, a shortened version of the title is used.


- Backup and Recovery 


This guide contains a subset of the information found in the Backup and Recovery book. The manual contains information about planning a backup and recovery strategy, the different types of media available to save and restore procedures, and disk recovery procedures. It also describes how to install the system again from backup.
- File Management
This guide provides information about using files in application programs.
- DB2 UDB for iSeries Database Programming
This guide provides a detailed description of the DB2 UDB for iSeries database organization, including information on how to create, describe, and update database files on the system.
- CL Programming 


This guide provides a wide-ranging discussion of the DB2 UDB for iSeries programming topics, including a general discussion of objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- Control Language (CL)
This guide provides a description of the DB2 UDB for iSeries control language (CL) and its OS/400 commands. (Non-OS/400 commands are described in the respective licensed program publications.) It also provides an overview of all the CL commands for the server, and it describes the syntax rules needed to code them.
- iSeries Security Reference 


This guide provides information about system security concepts, planning for security, and setting up security on the system. It also gives information about protecting the system and data from being used by people who do not have the proper authorization, protecting the data from intentional or unintentional damage or destruction, keeping security up-to-date, and setting up security on the system.
- DB2 UDB for iSeries SQL Reference
This guide provides information about DB2 UDB for iSeries statements and their parameters. It also includes an appendix describing the SQL communications area (SQLCA) and SQL description area (SQLDA).
- IDDU Use 

This guide describes how to use DB2 UDB for iSeries interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system.
- WebSphere Development Studio: ILE COBOL Programmer's Guide 


This guide provides information you need to design, write, test, and maintain COBOL for iSeries programs on the iSeries system.
- WebSphere Development Studio: ILE RPG Programmer's Guide 


This guide provides information you need to design, write, test, and maintain ILE RPG for iSeries programs on the iSeries system.
- ILE C for AS/400 Language Reference 

This guide provides information you need to design, write, test, and maintain ILE C for iSeries programs on the iSeries system.
- WebSphere Development Studio: ILE C/C++ Programmer's Guide 

This guide provides information you need to design, write, test, and maintain ILE C for iSeries programs on the iSeries system.
- WebSphere Development Studio: ILE COBOL Reference 

This guide provides information you need to design, write, test, and maintain COBOL for iSeries programs on the iSeriesCOBOL system.

- REXX/400 Programmer's Guide 

This guide provides information you need to design, write, test, and maintain REXX/400 programs on the iSeries system.
- DB2 Multisystem
This guide describes the fundamental concepts of distributed relational database files, nodegroups, and partitioning. The book provides the information you need to create and use database files that are partitioned across multiple iSeries systems. Information is provided on how to configure the systems, how to create the files, and how the files can be used in applications.
- Performance Tools for iSeries 

This guide provides the programmer with the information needed to collect data about the system, job, or program performance. This book also has tips for printing and analyzing performance data to identify and correct inefficiencies that might exist. Information about the manager and agent feature is included.
- DB2 UDB for iSeries SQL Call Level Interface (ODBC)
This guide provides the information necessary for application programmers to write applications using the DB2 call level interface.
- IBM Developer Kit for Java
This guide provides information you need to design, write, test, and maintain Java programs on the iSeries system. It also contains a chapter, iSeries Developer Kit for Java JDBC driver, which provides information about accessing database files from Java programs by using JDBC or SQLj.

Index

A

access plan
 definition 12
 in a package 12
 in a program 12
activation groups
 connection management
 example 334
aggregating function
 See UDFs (User-defined functions)
alias
 creating using iSeries Navigator 278
 definition 7
ALIAS names
 creating 55
ALTER TABLE statement 52
 adding a column 53
 changing a column 53
 check constraints 135
 constraints
 example removing 138
 data types
 allowable conversions 53
 deleting a column 54
 order of operation 54
 referential constraints 137, 138
AND keyword 77
 multiple search condition 77
API
 QSQCHKs 2
 QSQPRCED 2
application
 creating program 10
 design
 user-defined function (UDF) 231
 designing
 test data structure 323
 dynamic SQL
 designing and running 253
 overview 251
 establishing test environment 323
 performance verification 325
 program debugging 324
 program objects 10
 module 12
 output source file member 11
 program 12
 service program 13
 SQL package 12
 user source file member 11
SQLCODEs 324
SQLSTATEs 324
testing
 authorization 324
 input data 323
 programs 324
 testing SQL statements in
 program 323
application domain and
 object-orientation 189
application requester 327

application requester driver (ARD)
 programs
 package creation 350
 running statements 350
application server 327
ARD (application requester driver)
 programs
 See application requester driver (ARD)
 programs
atomic operation
 data definition statements (DDL) 316
 data integrity 316
 definition 316
auditing
 C2 security 306
authorization
 adding a user or group
 using iSeries Navigator 307
 Create SQL Package (CRTSQLPKG)
 command 330
 default public authority using iSeries
 Navigator 307
 for creating package 330
 for running using a package 330
 public authority using iSeries
 Navigator 307
 removing authority from objects
 iSeries Navigator 307
 testing 323, 324
auxiliary storage pools 310
 independent 321
 user 320

B

BETWEEN keyword 75
Binary Large Objects
 See BLOBs (Binary Large Objects)
BLOBs (Binary Large Objects)
 uses and definition 190

C

C2 security
 auditing 306
call level interface 2
CALL statement
 stored procedure 167, 168
 dynamic CALL 170
 example 168
 with SQLDA 169
call-type, passing to UDF 237
castability 202
casting, UDFs 214
catalog
 database design, use in 58
 definition 7
 getting information about 58
 column 58
 integrity 320

catalog (*continued*)
 LABEL ON information 51
 QSYS2 views 7
 table 58
CCSID
 connection to non-DB2 UDB for
 iSeries 333
 delimited identifier effect 333
 dynamic SQL statement 254
 package considerations 333
Change Class (CHGCLS) command 309
Change Job (CHGJOB) command 309
Change Logical File (CHGLF)
 command 309
Change Physical File (CHGPF)
 command 309
Character Large Objects
 See CLOBs (Character Large Objects)
check pending 144
 data integrity 318
clause
 DROP COLUMN 54
 FROM
 example 62
 GROUP BY
 example 65
 HAVING
 example 67
 INTO 94
 PREPARE statement, use with in
 dynamic SQL 257
 restrictions in dynamic SQL 262
 NULL value 71
 ORDER BY
 example 68
 SET 98
 column name 98
 constant 98
 DEFAULT 98
 expression 98
 host variable 98
 NULL value 98
 scalar subselect 98
 special register 98
 USING DESCRIPTOR 267
 WHENEVER NOT FOUND 126
WHERE
 column name 63
 comparison operators 65
 constant 64
 dynamic SQL example 267
 example 63
 expression 63
 host variable 64
 joining tables 79
 multiple search condition
 within 77
 NOT keyword 65
 NULL value 64
 special register 64
 subquery 64

- clause (*continued*)
 - WHERE CURRENT OF 127
- CLI 2
- CLOBs (Character Large Objects)
 - uses and definition 190
- CLOSECUR parameter
 - effect on implicit disconnect 338
- column
 - adding 53
 - changing definition 53
 - copying definitions using iSeries Navigator 38
 - defining heading 18, 51
 - defining using iSeries Navigator 36
 - definition 3, 7
 - deleting 54
 - FOR UPDATE OF clause 125
 - getting catalog information about 58
- column function
 - See UDFs (User-defined functions)
- command (CL)
 - Change Class (CHGCLS) 309
 - Change Job (CHGJOB) 309
 - Change Logical File (CHGLF) 309
 - Change Physical File (CHGPF) 309
 - CHGCLS (Change Class) 309
 - CHGJOB (Change Job) 309
 - CHGLF (Change Logical File) 309
 - CHGPF (Change Physical File) 309
 - Create Duplicate Object (CRTDUPOBJ) 324
 - Create SQL Package (CRTSQLPKG) 329, 375
 - Create User Profile (CRTUSRPRF) 306
 - CRTDUPOBJ (Create Duplicate Object)
 - command 324
 - CRTUSRPRF (Create User Profile) 306
 - Delete Library (DLTLIB) 317
 - Delete SQL Package (DLTSQLPKG) 330, 379
 - DLTLIB (Delete Library) 317
 - Edit Check Pending Constraints (EDTCCPST) 318
 - Edit Rebuild of Access Paths (EDTRBDAP) 318
 - Edit Recovery for Access Paths (EDTRCYAP) 319
 - EDTCCPST (Edit Check Pending Constraints) 318
 - EDTRBDAP (Edit Rebuild of Access Paths) 318
 - EDTRCYAP (Edit Recovery for Access Paths) 319
 - Grant Object Authority (GRTOBJAUT) 305
 - GRTOBJAUT (Grant Object Authority) 305, 309
 - Override Database File (OVRDBF) 128, 309
 - OVRDBF (Override Database File) 128, 309
 - Print SQL Information (PRTSQLINF) 380
 - Reclaim DDM connections (RCLDDMCNV) 346
- command (CL) (*continued*)
 - Revoke Object Authority (RVKOBJAUT) 305
 - Run SQL Statements (RUNSQLSTM) 2
 - RUNSQLSTM
 - errors 300
 - RUNSQLSTM (Run SQL statements) 2
 - RUNSQLSTM (Run SQL Statements) 299, 381
 - RVKOBJAUT (Revoke Object Authority) 305
 - Start Commitment Control (STRCMTCTL) 311
 - Start Journal Access Path (STRJRNAP) 319
 - STRCMTCTL (Start Commitment Control) 311
 - STRJRNAP (Start Journal Access Path) 319
 - STRSQL (Start SQL) 391
 - COMMENT ON statement
 - using, example 52
 - COMMIT
 - keyword 311
 - prepared statements
 - in dynamic SQL 255
 - statement 331
 - statement description 7
 - COMMIT statement 311
 - commitment control
 - activation group
 - example 334
 - committable updates 340
 - description 311
 - distributed connection
 - restrictions 343
 - DRDA resource 340
 - INSERT statement 95
 - job-level commitment definition 337, 343
 - protected resource 340
 - rollback required 345
 - RUNSQLSTM command 300
 - SQL statement processor 300
 - sync point manager 340
 - two-phase commit 340
 - unprotected resource 340
 - comparison operators 65
 - concurrency
 - data 308
 - deadlock detection 309
 - definition 308
 - CONNECT statement 328, 331
 - interactive SQL 296
 - connection
 - DDM 346
 - determining type 340
 - ending DDM 346
 - protected 340
 - unprotected 340
 - connection management
 - ARD programs 350
 - commitment control restrictions 343
 - distributed unit of work
 - considerations 345
- connection management (*continued*)
 - ending connections
 - DDMCNV effect on 346
 - DISCONNECT statement 346
 - RELEASE statement 346
 - example 334
 - implicit connection
 - default activation group 338
 - nondefault activation group 339
 - implicit disconnection
 - default activation group 338
 - nondefault activation group 339
 - multiple connections to same relational database 337
- connection status
 - determining 343
 - example 348
- consistency token 332
- consistent behavior and UDTs 215
- constraint
 - and sort sequence 119
 - check
 - adding 135
 - adding using iSeries Navigator 280
 - using 135
 - data integrity 317
 - definition 8
 - example
 - removing 138
 - iSeries Navigator
 - removing 282
 - key
 - adding using iSeries Navigator 279
 - referential 8
 - adding using iSeries Navigator 281
 - check pending 144
 - creating tables 137
 - delete rules 141
 - delete rules example 142
 - deleting from tables 141
 - inserting into tables 138
 - removing 138
 - update rules 140
 - updating tables 140
 - unique 8
 - UPDATE rules example 140
- correlation
 - definition 106
 - example 25
 - subqueries 109
 - example DELETE statement 113
 - example HAVING clause 111
 - example select list 112
 - example UPDATE statement 112
 - example WHERE clause 110
 - names 109
 - references 109
 - using subqueries 106
- CREATE ALIAS statement
 - creating and using 55
- CREATE DISTINCT TYPE statement
 - See also UDTs (User-defined types) and castability 202
 - examples of using 216

- CREATE DISTINCT TYPE statement
(*continued*)
 - to define a UDT 216
 - Create Duplicate Object (CRTDUPOBJ)
 - command 324
 - CREATE FUNCTION statement 238
 - See also* UDFs (User-defined functions)
 - AVG over a UDT example 208
 - BLOB string search example 207
 - counter for UDF 244
 - counting example 209
 - exponentiation example 206
 - external function with UDT parameter
 - example 208
 - save and restore considerations 206
 - search string over UDT example 208
 - square of a number UDF 242
 - string search example 207
 - table function example 209
 - to register a UDF 205
 - weather table UDF 244
 - CREATE INDEX statement
 - example 57
 - sort sequence 119
 - CREATE PROCEDURE statement 159
 - debugging 166
 - defining external 160
 - defining SQL 161
 - invoking 167
 - CREATE SCHEMA statement 16, 47
 - SQL statement processor 300
 - Create SQL Package (CRTSQLPKG)
 - command 329, 375
 - authority required 330
 - CREATE TABLE statement 48
 - AS 49
 - check constraints 135
 - constraints
 - example removing 138
 - default value 16
 - defining tables with UDTs 217
 - examples of using 217
 - identity columns
 - creating 50
 - removing 50
 - LIKE 48
 - NULL value 16
 - prompting
 - interactive SQL 289
 - referential constraints 137
 - ROWID 51
 - CREATE TRIGGER statement 148
 - AFTER trigger
 - example 150
 - BEFORE trigger
 - example 149
 - creating 148
 - handlers 151
 - transition tables 152
 - Create User Profile (CRTUSRPRF)
 - command 306
 - CREATE VIEW statement 55
 - description 30
 - example 31
 - read-only 56
 - using UNION 57
 - CREATE VIEW statement (*continued*)
 - WITH CASCADED CHECK OPTION 145
 - WITH CHECK OPTION 144
 - WITH LOCAL CHECK OPTION 146
 - cross join 81
 - CRTDUPOBJ (Create Duplicate Object)
 - command 324
 - CRTSQLPKG (Create SQL Package)
 - command 375
 - CRTUSRPRF command
 - create user profile 306
 - ctr() UDF C program listing 244
 - CURRENT DATE special register 71
 - CURRENT SCHEMA special register 71
 - CURRENT SERVER special register 71
 - CURRENT TIME special register 71
 - CURRENT TIMESTAMP special register 71
 - CURRENT TIMEZONE special register 71
 - cursor
 - closing
 - example 128
 - defining a cursor
 - example 124
 - delete current row
 - example 127
 - distributed unit of work 349
 - end-of-data
 - example 126
 - establishing position at end of table 122
 - example overview 122
 - open during a unit of work 133
 - open effect of recovery on 133
 - opening a cursor
 - example 126
 - retrieving a row
 - example 126
 - retrieving SELECT statement result
 - dynamic SQL 266
 - scrollable 122
 - serial 121
 - update current row
 - example 127
 - using 121
 - WITH HOLD clause 133
- ## D
- damage tolerance 319
 - data
 - committable updates 340
 - protection 305
 - viewing
 - using iSeries Navigator 40
 - data definition statements (DDL) 4, 47
 - atomic operation 316
 - data integrity 316
 - data dictionary
 - WITH DATA DICTIONARY clause
 - CREATE SCHEMA statement 6
 - data integrity 135
 - atomic operation 316
 - catalog 320
 - commitment control 311
 - data integrity (*continued*)
 - concurrency 308
 - deadlock detection 309
 - constraint 317
 - damage tolerance 319
 - data definition statements (DDL) 316
 - function 308
 - independent auxiliary storage pool (IASP) 321
 - index recovery 319
 - journaling 310
 - save/restore 318
 - savepoint 314
 - user auxiliary storage pool (ASP) 320
 - data manipulation statement (DML) 4, 61, 93
 - data types
 - allowable conversions 53
 - BLOBs 190
 - casting 72
 - CLOBs 190
 - DataLinks 225
 - commands used 227
 - FILE LINK CONTROL (database permissions) 227
 - FILE LINK CONTROL (file system permissions) 227
 - NO LINK CONTROL 227
 - DBCLOBs 190
 - object-oriented 189
 - user-defined
 - See* UDFs (User-defined functions)
 - database
 - design, using the catalog in 58
 - relational 3
 - Database Navigator 271
 - adding new objects to a map 273
 - changing objects in a map 273
 - creating
 - user-defined relationship 273
 - creating a map 272
 - DataLinks 225
 - commands used 227
 - FILE LINK CONTROL
 - (database permissions) 227
 - file system permissions 227
 - NO LINK CONTROL 227
 - date format 73
 - date/time arithmetic 74
 - DB2 Multisystem 2
 - DB2 Query Manager for iSeries 2
 - DB2 UDB for iSeries 1
 - See also* Structured Query Language
 - considerations for packages 330
 - distributed relational database support 328
 - DB2 UDB for iSeries sample table 353
 - DB2 UDB Query Manager and SQL Development Kit 1
 - distributed relational database support 328
 - DB2 UDB Symmetric Multiprocessing 3
 - DB2 Universal Database for iSeries
 - See* DB2 UDB for iSeries
 - DBCLOBs (Double-Byte Character Large Objects)
 - uses and definition 190

- DBCS (double-byte character set)
 - considerations in interactive SQL 290
 - DBGVIEW(*SOURCE) parameter 325
 - DBINFO keyword
 - functions 238
 - deadlock detection 309
 - DECLARE CURSOR statement
 - using 70
 - DECLARE GLOBAL TEMPORARY TABLE statement 49
 - default value
 - inserting in a view 56
 - Delete Library (DLTLIB) command 317
 - Delete SQL Package (DLTSQLPKG) command 330, 379
 - DELETE statement 93
 - correlated subquery, use in 113
 - delete rules 141
 - delete rules example 142
 - deleting from tables 141
 - description 29, 102
 - example 29
 - derived table 83
 - DESCRIBE statement
 - use with dynamic SQL 255
 - DESCRIBE TABLE statement 331
 - DFT_SQLMATHWARN configuration parameter 239
 - DISCONNECT statement 328, 331
 - ending connection 346
 - distinct type
 - See UDTs (User-defined types)
 - distributed relational database
 - accessing remote databases 295
 - application requester 327
 - application server 327
 - committable updates 340, 343
 - connection management 334
 - multiple connections 337
 - connection restrictions 343
 - connection type
 - determining 340
 - protected 340
 - unprotected 340
 - consideration for creating packages 330
 - creating packages 330
 - DB2 UDB for iSeries support 328
 - determining connection status 343
 - distributed RUW example program 328
 - distributed unit of work 327, 339, 346
 - ending connections
 - DDMCNV effect on 346
 - DISCONNECT statement 346
 - RELEASE statement 346
 - first failure data capture (FFDC) 351
 - implicit connection
 - default activation group 338
 - nondefault activation group 339
 - implicit disconnection
 - default activation group 338
 - nondefault activation group 339
 - interactive SQL 295
 - packages 329
 - statement in 330
 - distributed relational database (*continued*)
 - precompiler diagnostic messages 330
 - problem handling 351
 - protected connection 340
 - protected resource 340
 - remote unit of work 327, 339
 - rollback required state 345
 - session attributes 296
 - SQL packages 329
 - stored procedure considerations 351
 - sync point manager 340
 - two-phase commit 340
 - unprotected connection 340
 - unprotected resource 340
 - valid SQL statements 330
 - Distributed Relational Database Architecture (DRDA) 1
 - distributed unit of work 327, 339, 346
 - connection considerations 345
 - connection status 343
 - connection type 340
 - cursors 349
 - managing connections 347
 - prepared statements 349
 - sample program 347
 - DLTSQLPKG (Delete SQL Package) command 379
 - Double-Byte Character Large Objects
 - See DBCLOBs (Double-Byte Character Large Objects)
 - DRDA (Distributed Relational Database Architecture)
 - See Distributed Relational Database Architecture (DRDA)
 - DRDA level 1
 - See remote unit of work
 - DRDA level 2
 - See distributed unit of work
 - DRDA resource 340
 - DROP COLUMN clause
 - example 54
 - DROP PACKAGE statement 328
 - DROP statement 59
 - dropping an alias 55
 - DUW (distributed unit of work)
 - See distributed unit of work
 - dynamic SQL
 - address variable 251
 - allocating storage 257
 - allocating storage for SQLDA 262
 - application 251, 253
 - assignments of UDTs 221
 - building and running statements 251
 - CCSID 254
 - cursor, use in 256
 - DESCRIBE statement 255
 - example of allocating storage for SQLDA 262
 - fixed-list SELECT statement 255
 - parameter marker 267
 - processing non-SELECT statements 254
 - replacing parameter markers with host variables 267
 - run-time overhead 251
 - SELECT statement result cursor, using 266
 - dynamic SQL (*continued*)
 - SQLDA (SQL descriptor area) 257
 - SQLDA (SQL descriptor area) format 257
 - statements 4
 - using EXECUTE statement 254
 - using PREPARE statement 254
 - varying-list SELECT statement 255, 256
- ## E
- Edit Check Pending Constraints (EDTIPCST) command 318
 - Edit Rebuild of Access Paths (EDTRBDAP) command 318
 - Edit Recovery for Access Paths (EDTRCYAP) command 319
 - encapsulation and UDTs 215
 - error determination
 - in distributed relational database
 - first failure data capture (FFDC) 351
 - examples
 - adding constraints 137
 - AFTER trigger 150
 - assignments in dynamic SQL 221
 - assignments involving different UDTs 221
 - assignments involving UDTs 221
 - AVG over a UDT 208
 - BEFORE trigger 149
 - BETWEEN 75
 - casting between UDTs 219
 - catalog
 - getting column information 58
 - getting table information 58
 - changing data
 - SET clause 98
 - with host variables 98
 - changing information in a table 27
 - changing information in a table using iSeries Navigator 40
 - changing rows in table
 - host variables 98
 - check constraints 135
 - closing a cursor 128
 - COMMENT ON statement 52
 - comparisons involving UDTs 218, 220
 - connection management 334
 - copying a table
 - using iSeries Navigator 41
 - copying column definitions
 - using iSeries Navigator 38
 - correlated subquery
 - DELETE statement 113
 - HAVING clause 111
 - select list 112
 - UPDATE statement 112
 - WHERE clause 110
 - correlation name 25
 - cost of a UDT 208
 - counter for UDFs 244
 - counting and defining UDFs 209
 - CREATE ALIAS statement 55
 - CREATE SCHEMA statement 47

examples (continued)

CREATE TABLE AS statement 49
 CREATE TABLE LIKE statement 48
 CREATE TABLE statement 48
 CREATE VIEW 55
 CREATE VIEW statement
 view on a table 31
 creating
 index 57
 schema 16
 table 17
 table in iSeries Navigator 36
 creating identity columns 50
 creating library with iSeries
 Navigator 34
 creating view on a table using iSeries
 Navigator 42, 44
 CROSS JOIN 81
 ctr() UDF C program listing 244
 CURRENT DATE 73
 CURRENT TIMEZONE 74
 cursor 122
 cursor in DUW program 349
 debugging stored procedure 166
 DECLARE GLOBAL TEMPORARY
 TABLE statement 49
 defining a cursor 124
 defining columns using iSeries
 Navigator 36
 defining stored procedures
 with CREATE PROCEDURE 160,
 161
 defining tables with UDTs 217
 defining the UDT and UDFs 223
 DELETE
 from table 102
 delete current row 127
 delete rules example 142
 deleting information from a table
 using iSeries Navigator 40
 deleting information in a table 29
 deleting library using iSeries
 Navigator 46
 deleting schema using iSeries
 Navigator 46
 deleting table using iSeries
 Navigator 46
 deleting view using iSeries
 Navigator 46
 determining connection status 348
 distributed RUW program 328
 distributed unit of work
 program 347
 DROP statement 59
 dynamic CALL 170
 stored procedure 170
 edit list of libraries displayed
 using iSeries Navigator 35
 embedded CALL 167, 168
 with SQLDA 169
 end-of-data 126
 EXCEPTION JOIN 81
 EXISTS 75
 exponentiation and defining
 UDFs 206
 external trigger 152

examples (continued)

extracting a document to a file (CLOB
 elements in a table) 196
 function invocations 211
 getting comment 52
 IN 75
 INNER JOIN 79
 using WHERE 79
 INSERT statement 20
 inserting into identity columns 97
 inserting
 row to table 94
 inserting data into a CLOB
 column 198
 inserting data with constraints 139
 inserting information into a table
 using iSeries Navigator 38
 inserting multiple rows
 using blocked INSERT 96
 using SELECT 95
 invoking stored procedures 168, 170
 where a CREATE PROCEDURE
 exists 167
 where no CREATE PROCEDURE
 exists 168
 IS NULL 75
 JOIN clause 25
 LABEL ON statement 19, 51
 LEFT OUTER JOIN 80
 LIKE 75
 list function in interactive SQL 291
 LOB function to populate the
 database 224
 LOB locators to manipulate UDT
 instances 225
 LOBFILE.SQB COBOL program
 listing 198
 LOBFILE.SQC C program listing 197
 LOBLOC.SQB COBOL program
 listing 194
 LOBLOC.SQC C program listing 192
 moving a table
 using iSeries Navigator 41
 multiple join types in one
 statement 82
 multiple search condition (WHERE
 clause) 77
 opening a cursor 126
 ORDER BY
 sort sequence 116
 parameter markers in functions 211
 preventing duplicate rows 74
 QSYSPRT listing
 SQL statement processor 301
 removing constraints 138
 removing identity columns 50
 retrieving a row 126
 returning a table function 209
 RIGHT OUTER JOIN 80
 ROWID 51
 sample table 353
 search string and BLOBs 207
 SELECT rows
 sort sequence 117
 SELECT statement 22, 61
 performing complex search
 condition 75

examples (continued)

SELECT statement allocating storage
 for SQLDA 262
 simulating a full outer join 82
 special register 73, 74
 square of a number UDF 242
 stored procedures
 returning completion status 179
 returning completion status ILE C
 and PL/I 179
 returning completion status
 REXX 184
 string search and defining UDFs 207
 string search over UDT 208
 subquery
 basic comparisons 107
 comparisons 107
 EXISTS 108
 IN 108
 subquery in SELECT 105
 table
 ACT 366
 CL_SCHED 367
 DEPARTMENT 354
 EMP_PHOTO 357
 EMP_RESUME 358
 EMPLOYEE 355
 EMPPROJECT 359
 IN_TRAY 368
 ORG 369
 PROJECT 364
 PROJECT 362
 SALES 371
 STAFF 370
 UDFs to query instances of
 UDTs 224
 UNION
 simple 88
 using host variables 85
 UNION ALL 88
 unqualified function reference 212
 update current row 127
 UPDATE rules for constraints 140
 UPDATE statement 27
 as data is retrieved 100
 identity column 100
 scalar subselect 99
 using SELECT 99
 use of UDTs in UNION 222
 user-defined sourced functions on
 UDTs 220
 using a locator to work with a CLOB
 value 192
 using CREATE DISTINCT TYPE 216
 using qualified function
 reference 212
 using table expressions 83
 view
 sort sequence 118
 view WITH CASCADED CHECK
 OPTION 146
 view WITH LOCAL CHECK
 OPTION 146
 viewing contents of a table or view
 using iSeries Navigator 40
 views over multiple tables 31
 weather table UDF 244

examples (*continued*)
WHERE clause
AND 77
OR 77
exception join 81
EXECUTE privileges
for packages 330
EXECUTE statement
in dynamic SQL 254
EXISTS keyword 75
use in subquery 108
extended dynamic
QSQRCEd 2
extensibility and UDTs 215

F

FETCH statement 128
dynamic SQL 266
using descriptor area 131
using host structure array 129
using row storage area 131
FFDC (first failure data capture)
See first failure data capture (FFDC)
field
definition 3
file reference variables
examples of using 196
input values 195
output values 196
first failure data capture (FFDC) 351
flexibility and UDTs 215
FOR UPDATE OF clause
restrictions 125
FROM clause
description 62
function
See also UDFs (User-defined functions)
invocations example 211
references, summary for UDFs 213
syntax for referring to 210, 211

G

generate SQL 277
editing list of objects 277
Grant Object Authority (GRTOBJAUT)
command 305
GRANT PACKAGE statement 328
GROUP BY
clause 65
using NULL value with 66

H

HAVING
clause 67

I

IDDU (interactive data definition
utility) 7
identity column
creating 50
inserting into 97

identity column (*continued*)
removing 50
ILE programs
package 332
ILE service programs
package 332
implicit connect
See connection management
implicit disconnect
See connection management
IN keyword
description 75
subquery, use in 108
independent auxiliary storage pool
(IASP) 321
index
add 57
adding using iSeries Navigator 278
definition 8
iSeries Navigator
removing 282
journaling 319
rebuild 319
recovery 319
save and restore 319
indicator variables
and LOB locators 195
stored procedures 176
infix notation and UDFs 213
inner join 79
INSERT statement 93
and referential constraints 138
blocked 93
commitment control 95
default value 20, 94
description 93
example 20
inserting constant 94
inserting data with constraints
example 139
inserting DEFAULT 94
inserting expression 94
inserting host variable 94
inserting identity column 97
inserting in a view 56
inserting into alias 55
inserting multiple rows
using blocked INSERT 96
using SELECT 95
inserting NULL 94
inserting special register 94
inserting subquery 94
INTO clause 94
NULL value 94
reusing deleted rows 95
VALUES clause 93
Integrated Language Environment (ILE)
module 12
program 12
service program 13
interactive data definition utility
See IDDU
interactive interface
concepts 2
interactive SQL 2
accessing remote databases 295
adding DBCS data 290

interactive SQL (*continued*)
change session attributes 293
description 285
exiting 294
function 285
general use 285
getting started 286
list selection function 290
overview 285
package 296
printing current session 294
prompting 288
DBCS consideration 290
overview 285
prompting subquery 289
recovering an SQL session 295
removing all entries from current
session 294
saving session 294
session services 286, 293
statement entry 285, 287
statement processing mode 289
syntax checking 289
terminology 3
testing your SQL statements
with 285
using an existing session 295
INTO clause
description 94
PREPARE statements
in dynamic SQL 257
restriction
dynamic SQL 262
IS NULL keyword 75
iSeries Navigator 33
adding
referential constraint 281
advanced functions 271
advanced functions for tables 277
alias
creating 278
authorizing
users and groups 307
changing information in a table 40
check constraint
adding 280
constraint
removing 282
copying a table 41
creating a library 33
creating a schema 33
creating a table 35
example 36
creating a view 42
Database Navigator 271
adding new objects to a map 273
changing the objects in a
map 273
creating a map 272
creating a user-defined
relationship 273
deleting information from a table 40
edit list of libraries displayed 35
generate SQL 277
edit list of objects 277
index
adding 278

- iSeries Navigator (*continued*)
 - index (*continued*)
 - removing 282
 - key constraint
 - adding 279
 - moving a table 41
 - package
 - creating 284
 - public authority 307
 - setting up default 307
 - removing authority 307
 - Run SQL Scripts 274
 - changing the options 275
 - creating a script 275
 - running a script 275
 - viewing the job log 276
 - viewing the result set for a procedure 276
 - securing data 307
 - stored procedures
 - defining 283
 - trigger
 - adding 281
 - disabling 282
 - enabling 282
 - removing 282
 - UDFs (User-defined functions)
 - defining 283
 - UDTs (User-defined types)
 - defining 284
 - viewing
 - contents of a table or view 40

J

- job attribute
 - DDMCNV 346
- job-level commitment definition 337, 343
- join
 - data from multiple tables 78
- JOIN clause
 - definition 25
- journal
 - definition 7
- journal receiver
 - definition 7
- journaling 310

K

- keyword
 - AND 77
 - BETWEEN 75
 - COMMIT 311
 - DISTINCT 74
 - EXISTS 75, 108
 - IN 75
 - IS NULL 75
 - LIKE 75
 - considerations 76
 - NOT 65, 77
 - OR 77
 - UNION 85
 - UNION ALL 88

L

- LABEL ON statement 18, 51
 - information in catalog 51
 - package 332
- left outer join 80
- library
 - creating with iSeries Navigator 33
 - definition 3
 - deleting using iSeries Navigator 46
 - edit list displayed in iSeries Navigator 35
- LIKE keyword 75
 - considerations 76
- LOBs (Large Objects)
 - and DB2 object extensions 189
 - control information to access large object data 191
 - display layout of columns 199
 - file reference variables 190
 - examples of using 196
 - input values 195
 - output values 196
 - SQL_FILE_APPEND, output value option 196
 - SQL_FILE_CREATE, output value option 196
 - SQL_FILE_OVERWRITE, output value option 196
 - SQL_FILE_READ, input value option 196
 - journal entry layout 199
 - large object descriptor 190
 - large object value 190
 - LOB function to populate the database example 224
 - LOB locators to manipulate UDT instances example 225
 - LOBEVAL.SQB COBOL program listing 198
 - LOBEVAL.SQC C program listing 197
 - LOBLOC.SQB COBOL program listing 194
 - LOBLOC.SQC C program listing 192
 - locators 190, 191
 - example of using 192
 - indicator variables 195
 - manipulating 189
 - maximum size for large object columns, defining 191
 - programming options for values 191
 - storing 189
 - synergy with UDTs and UDFs
 - examples of complex applications 223
- LOCK TABLE statement 309
- logical file 7
 - definition 3
- LONG VARCHAR
 - storage limits 190
- LONG VARGRAPHIC
 - storage limits 190
- Loosely Coupled Parallelism 2

M

- member
 - output source file 11
- mode
 - interactive SQL 289
- module
 - Integrated Language Environment (ILE)
 - object 12

N

- naming convention
 - *SQL 3
 - *SYS 3
 - SQL 4
 - system 4
- NOT keyword 65, 77
 - multiple search condition 77
- NULL value 71
 - INSERT INTO clause, value 94
 - INSERT statement 94
 - inserting in a view 56
 - SET clause, value 98
 - UPDATE statement 98
 - used with GROUP BY clause 66
 - used with ORDER BY clause 69
 - WHERE clause 64

O

- object-orientation and UDFs 201
- object-oriented extensions and UDTs 215
- object-relational
 - application domain and object-orientation 189
 - constraint mechanisms 189
 - data types 189
 - definition 189
 - LOBs 189
 - support for 190
 - triggers 189
 - UDTs and UDFs 189
 - why use the DB2 object extensions 189
- OPEN statement 267
- operators, comparison 65
- OR keyword 77
 - multiple search condition 77
- ORDER BY
 - clause 68
 - using NULL values with 69
 - sort sequence, using 115
 - with sort sequence 116
- output source file member
 - definition 11
- Override Database File (OVRDBF)
 - command 128, 309

P

- package
 - authority to create 330
 - authority to run 330
 - bind to an application 9

- package (*continued*)
 - CCSID considerations for 333
 - consistency token 332
 - Create SQL Package (CRTSQLPKG)
 - command 329
 - authority required 330
 - creating
 - authority required 330
 - effect of ARD programs 350
 - errors during 330
 - on local system 332
 - RDB parameter 329
 - RDBCNNMTH parameter 332
 - TGTRLS parameter 331
 - type of connection 332
 - unit of work boundary 332
 - creating on a non-DB2 UDB for iSeries
 - errors during 330
 - required precompiler options for DB2 Common Server 330
 - unsupported precompiler options 330
 - DB2 UDB for iSeries support 329
 - definition 9, 12, 329
 - Delete SQL Package (DLTSQLPKG)
 - command 330
 - deleting 330
 - interactive SQL 296
 - iSeries Navigator
 - creating 284
 - labeling 332
 - object type 332
 - restore 332
 - save 332
 - SQL statement size 331
 - statements that do not require package 331
- parameter markers
 - in dynamic SQL 267
 - in functions example 211
- parameter passing
 - stored procedures 171, 176
- performance
 - and UDTs 215
 - UDFs 201
 - verification 325
- physical file 7
 - definition 3
- precompiler
 - concepts 1
 - diagnostic messages 330
- precompiler command
 - CRTSQLxxx 116, 330
- precompiler parameter
 - DBGVIEW(*SOURCE) 325
- PREPARE statement
 - in dynamic SQL 254
- prepared statement
 - distributed unit of work 349
- Print SQL Information (PRTSQLINF)
 - command 380
- program
 - application program
 - See* application
 - debugging 324
 - definition 12

- program (*continued*)
 - Integrated Language Environment (ILE) object 12
 - non-ILE object 12
 - performance verification 325
 - protected connections
 - dropping 343
 - protected resource 340
 - PRTSQLINF (Print SQL Information)
 - command 380
 - public authority 305
 - defining
 - using iSeries Navigator 307
 - setting up default
 - using iSeries Navigator 307

Q

- QSQCCHKS 2
- QSQPRCED 2
 - package 9
- QSYS2
 - catalog views 7
- QSYSVRT listing
 - SQL statement processor
 - example 301

R

- read-only
 - table
 - cursor 125
- read-only connection 340
- Reclaim DDM connections (RCLDDMCNV) command 346
- record
 - definition 3
- recursion
 - SQL 333
- referential constraints
 - inserting into tables 138
 - removing 138
- referential integrity 136
 - definition 8
- relational database 3
- RELEASE statement 328, 331
 - ending connection 346
- remote databases
 - accessing from interactive SQL 295
- remote unit of work 327, 339
 - connection status 343
 - connection type 340
 - example program 328
- restriction
 - FOR UPDATE OF 101
- Revoke Object Authority (RVKOBJAUT)
 - command 305
- REVOKE PACKAGE statement 328
- REXX 2
- right outer join 80
- rollback
 - rollback required state 345
- ROLLBACK statement 311, 331
 - prepared statements
 - in dynamic SQL 255

- row
 - definition 3, 7
 - inserting multiple using blocked INSERT
 - into a table 96
 - inserting multiple using SELECT
 - into a table 95
 - preventing duplicate 74
 - reusing deleted with INSERT 95
 - selection using sort sequence 115

- row selection
 - and sort sequence 117

ROWID

- using in a table 51
- RRN scalar function 80
- Run SQL Scripts 2, 274
 - changing the options 275
 - creating a script
 - script 275
 - running a script 275
 - viewing the job log 276
 - viewing the result set for a procedure 276
- Run SQL Statements (RUNSQLSTM)
 - command 2, 381
- run-time support
 - concepts 1
- RUNSQLSTM (Run SQL Statements)
 - command 2
 - command (CL) 299, 381
 - command errors 300
 - comment 300
 - commitment control 300
 - source file 300
- RUW (remote unit of work)
 - See* remote unit of work

S

- sample programs
 - distributed RUW program 328
- sample tables DB2 UDB for iSeries 353
 - ACT 366
 - CL_SCHED 367
 - DEPARTMENT 354
 - EMP_PHOTO 357
 - EMP_RESUME 358
 - EMPLOYEE 355
 - EMPPROJECT 359
 - IN_TRAY 368
 - ORG 369
 - PROJECT 364
 - PROJECT 362
 - SALES 371
 - STAFF 370
- save/restore 318
 - packages 332
- savepoint
 - data integrity 314
 - definition 314
- SAVEPOINT statement 314
 - considerations for distributed databases 316
 - levels 315

- scalar function
 - See UDFs (User-defined functions)
- schema
 - auxiliary storage pools 310
 - creating 16
 - creating with iSeries Navigator 33
 - creating with SQL 16
 - definition 3, 6
 - deleting using iSeries Navigator 46
 - SQL statement processor 300
- scrollable cursor
 - See cursor
- search condition
 - performing complex 75
- security 305
 - authorization 324
 - authorization ID 306
 - commitment control 311
 - data integrity 308
 - concurrency 308
 - public authority 305
 - using iSeries Navigator 307
 - view 306
- SELECT INTO statement 70
 - in dynamic SQL 253
- SELECT statement 61
 - AND keyword 77
 - example 77
 - asterisk (select all columns) 62
 - BETWEEN 75
 - casting data types 72
 - correlation name
 - example 25
 - CROSS JOIN 81
 - data retrieval errors 89
 - date value 73
 - date/time arithmetic 74
 - definition 22
 - DISTINCT keyword 74
 - dynamic SQL
 - retrieving SELECT statement result 266
 - example of allocating storage for SQLDA 262
 - EXCEPT JOIN 81
 - EXISTS 75
 - FROM clause 62
 - GROUP BY
 - example 65
 - using NULL value with 66
 - HAVING
 - example 67
 - IN 75
 - INNER JOIN 79
 - IS NULL 75
 - joining information
 - example 25
 - joins 78
 - LEFT OUTER JOIN 80
 - LIKE 75
 - considerations 76
 - multiple join types in one statement 82
 - NOT keyword 77
 - NULL value
 - example 71
 - OR keyword 77
- SELECT statement (*continued*)
 - example 77
 - ORDER BY
 - example 68
 - using NULL values with 69
 - performing complex search condition 75
 - preventing duplicate rows 74
 - processing and using SQLDA 255
 - RIGHT OUTER JOIN 80
 - simulating a full outer join 82
 - special register
 - example 71
 - specifying column 62
 - subquery
 - definition 105
 - example 105
 - time value 73
 - timestamp value 73
 - UNION 85
 - UNION ALL 88
 - using fixed-list 255
 - using table expressions 83
 - using varying-list 256
 - WHERE
 - multiple search conditions 77
- WHERE clause 63
 - column name 63
 - comparison operators 65
 - constant 64
 - expressions 63, 64
 - host variable 64
 - inner join 79
 - NOT keyword 65
 - NULL value 64
 - predicates 63
 - special register 64
 - subquery 64
- serial cursor
 - See cursor
- service program
 - Integrated Language Environment (ILE)
 - object 13
- session services
 - accessing remote databases 295
 - change session attributes in interactive SQL 293
 - exiting interactive SQL 294
 - in interactive SQL 293
 - printing current session in interactive SQL 294
 - recovering an SQL session 295
 - removing all entries from current session 294
 - saving session 294
 - using an existing session 295
- SET clause
 - column name 98
 - constant 98
 - DEFAULT 98
 - description 98
 - expression 98
 - host variable 98
 - NULL value 98
 - scalar subselect 98
 - special register 98
- SET CONNECTION statement 328, 331
- SET CURRENT FUNCTION PATH
 - statement 204
- SET TRANSACTION statement
 - effect on implicit disconnect 338
 - not allowed in package 330
- sort sequence
 - and constraints 119
 - and row selection 117
 - CREATE INDEX 119
 - used with ORDER BY 115
 - used with row selection 115
 - using 115
 - using with ORDER BY 116
 - views 118
- source file
 - member, output
 - definition 11
 - member, user 11
- sourced function
 - See UDFs (User-defined functions)
- sourced UDF
 - See UDFs (User-defined functions)
- special register
 - CURRENT DATE 71
 - CURRENT SCHEMA 71
 - CURRENT SERVER 71
 - CURRENT TIME 71
 - CURRENT TIMESTAMP 71
 - CURRENT TIMEZONE 71
 - USER 71
- SQL 1
 - call level interface 2
 - generate SQL 277
 - introduction 1
 - naming conventions 3
 - object descriptions 6
 - recursion 333
 - statements
 - types 4
- SQL Communication Area (SQLCA)
 - See SQLCA (SQL Communication Area)
- SQL descriptor area (SQLDA)
 - See SQLDA (SQL descriptor area)
- SQL naming convention 4
- SQL package
 - definition 3
- SQL statement processor
 - commitment control 300
 - example
 - QSYSPT listing 301
 - schemas 300
 - using 299
- SQL_FILE_READ, input value
 - option 196
- SQLCA (SQL Communication Area) 6
 - SQLERRD field 340, 343
 - SQLERRD(4)
 - determining connection status 343
 - determining connection type 340
- SQLCODEs
 - testing application program 324
- SQLDA (SQL descriptor area)
 - allocating storage for 262
 - format 257

- SQLDA (SQL descriptor area) *(continued)*
 - processing SELECT statement 255
 - programming language, use in 257
 - SELECT statement for allocating storage for SQLDA 262
 - SQLD 258
 - SQLDABC 258
 - SQLDAID 258
 - SQLDATA 258
 - SQLDATALEN 262
 - SQLDATATYPE_NAME 262
 - SQLIND 259
 - SQLLEN 258
 - SQLLONGLEN 262
 - SQLN 258
 - SQLNAME 259
 - SQLRES 258
 - SQLTYPE 258
 - SQLVAR 258
 - SQLVAR2 261
- SQLSTATES
 - testing application program 324
- Start Commitment Control (STRCMTCTL) command 311
- Start Journal Access Path (STRJRNP) command 319
- Start SQL (STRSQL) command 391
- statements
 - CALL 167, 168
 - dynamic with stored procedure 170
 - example 168
 - with SQLDA 169
 - COMMENT ON statement 52
 - COMMIT 7, 311
 - CONNECT 328
 - CREATE ALIAS statement
 - example 55
 - CREATE INDEX
 - sort sequence 119
 - CREATE PROCEDURE
 - debugging 166
 - defining external 160
 - defining SQL 161
 - external procedure 159
 - invoking 167
 - SQL procedure 159
 - CREATE SCHEMA 16
 - SQL statement processor 300
 - CREATE SCHEMA statement
 - example 47
 - CREATE TABLE 16
 - CREATE TABLE AS statement
 - example 49
 - CREATE TABLE LIKE statement
 - example 48
 - CREATE TABLE statement 48
 - CREATE VIEW 30, 55
 - example 31
 - data definition (DDL) 4
 - data manipulation (DML) 4
 - date value 73
 - date/time arithmetic 74
 - DECLARE CURSOR 70
 - DECLARE GLOBAL TEMPORARY TABLE statement
 - example 49
 - statements *(continued)*
 - DELETE 93
 - example 29, 102
 - DISCONNECT 328
 - DROP PACKAGE 328
 - dynamic 4
 - EXECUTE 254
 - FETCH
 - dynamic SQL 266
 - multiple-row 128
 - using descriptor area 131
 - using host structure array 129
 - using row storage area 131
 - GRANT PACKAGE 328
 - INSERT 93
 - INTO clause 94
 - using 93
 - LABEL ON statement
 - example 51
 - examples 18
 - LOCK TABLE 309
 - OPEN 267
 - package not required 331
 - packages 330
 - PREPARE
 - non-SELECT statement 254
 - processing non select 254
 - RELEASE 328
 - REVOKE PACKAGE 328
 - ROLLBACK 7, 311
 - SAVEPOINT 314, 315, 316
 - SELECT 22
 - AND keyword 77
 - BETWEEN 75
 - example 61
 - EXISTS 75
 - IN 75
 - IS NULL 75
 - LIKE 75
 - LIKE, considerations 76
 - NOT keyword 77
 - OR keyword 77
 - performing complex search condition 75
 - preventing duplicate rows 74
 - specifying column 62
 - WHERE, multiple search conditions 77
 - SELECT INTO
 - dynamic SQL 253
 - SET CONNECTION 328
 - SQL packages 330
 - testing
 - in application program 323
 - using interactive SQL 285
 - time value 73
 - timestamp value 73
 - UPDATE 93
 - changing data value 27
 - example 97
 - stored procedure
 - returning completion status 179
 - ILE C and PL/I 179
 - REXX 184
 - stored procedures 159
 - considerations in distributed relational database 351
- stored procedures *(continued)*
 - debugging 166
 - defining external 160
 - defining SQL 161
 - definition 9
 - dynamic CALL 170
 - embedded CALL
 - with SQLDA 169
 - invoking 167
 - invoking using CALL 167
 - invoking using embedded CALL 168
 - iSeries Navigator
 - defining 283
 - parameter passing 171
 - indicator variables 176
 - returning a completion status with SQLDA 178
 - strong typing and UDTs 218
 - STRSQL (Start SQL) command 286, 391
 - Structured Query Language 1
 - subquery 105
 - ALL 107
 - ANY 107
 - basic comparison 107
 - correlated 106, 109
 - example DELETE statement 113
 - example HAVING clause 111
 - example select list 112
 - example UPDATE statement 112
 - example WHERE clause 110
 - correlated names and references 109
 - definition 105
 - examples
 - in SELECT 105
 - EXISTS keyword 108
 - IN keyword 108
 - notes on using 108
 - prompting 289
 - quantified comparison 107
 - search condition 106
 - SOME 107
 - Symmetric Multiprocessing 3
 - sync point manager 340
 - syntax check
 - QSQCCHKS 2
 - syntax for referring to functions 210
 - system naming convention 4

T

- table
 - changing data
 - SET CLAUSE 98
 - with host variables 98
 - changing definition 52
 - changing information in 27
 - changing information using iSeries Navigator 40
 - copying
 - using iSeries Navigator 41
 - copying column definitions using iSeries Navigator 38
 - creating
 - CREATE TABLE statement 16, 17
 - using iSeries Navigator 35
 - creating view 31
 - CROSS JOIN 81

table (*continued*)

- DB2 UDB for iSeries sample 353
- defining columns using iSeries Navigator 36
- defining name 51
- definition 3, 7
- deleting information
 - using iSeries Navigator 40
- deleting information in 29
- deleting using iSeries Navigator 46
- derived 83
- establishing position at the end 122
- EXCEPTION JOIN 81
- getting catalog information
 - about column 58
- getting information 22
 - from multiple 25
- INNER JOIN 79
 - using WHERE 79
- inserting
 - information into 20
- inserting information into 38
- inserting multiple rows into
 - using blocked INSERT 96
 - using SELECT 95
- joining 78
- LEFT OUTER JOIN 80
- moving
 - using iSeries Navigator 41
- multiple join types in one statement 82
- RIGHT OUTER JOIN 80
- simulating a full outer join 82
- updating data 97
- used in examples
 - ACT 366
 - CL_SCHED 367
 - DEPARTMENT 354
 - EMP_PHOTO 357
 - EMP_RESUME 358
 - EMPLOYEE 355
 - EMPPROJECT 359
 - IN_TRAY 368
 - ORG 369
 - PROJECT 364
 - PROJECT 362
 - SALES 371
 - STAFF 370
- using 16, 17
 - with iSeries Navigator 35
- using table expressions 83
- view over multiple 31

table function

See UDFs (User-defined functions)

time format 73

timestamp format 73

trigger 147

- AFTER trigger
 - example 150
- and DB2 object extensions 189
- BEFORE trigger
 - example 149
- creating SQL 148
- definition 8
- external trigger 152
 - example 152
- handlers 151

trigger 147 (*continued*)

- iSeries Navigator
 - adding 281
 - disabling 282
 - enabling 282
 - removing 282
- SQL 148
- transition tables 152
- two-phase commit 340

U

UDFs (User-defined functions)

- aggregating functions 204
- and DB2 object extensions 189
- CAST FROM clause 234, 238, 239
- casting 214
- column functions 204
- compiling 205
- concepts 203
- defining the UDT and UDFs
 - example 223
- definition 9, 200
- fenced versus unfenced 242
- function path 203
- function selection algorithm 203
- general considerations 213
- implementing UDFs 200
- infix notation 213
- invoking
 - examples of invocations 210
 - parameter markers in functions 211
 - qualified function reference 212
 - unqualified function reference 212
- iSeries Navigator
 - defining 283
- length of time 231
- LOB types 214
- object-orientation 201
- overloaded function names 203
- parallel processing 232
- parameter style DB2GENERAL 240
- parameter style DB2SQL 236
- parameter style GENERAL 238
- parameter style GENERAL WITH NULLS 239
- parameter style JAVA 240
- parameter style SQL 234
- parameter style SIMPLE CALL 238
- passing argument with DBINFO 238
- passing call-type 237
- passing diagnostic-message 236
- passing function-name 235
- passing scratchpad 237
- passing specific-name 236
- passing SQL-argument 234, 238, 239
- passing SQL-argument-ind 235
- passing SQL-argument-ind-array 239
- passing SQL-result 234, 238, 239
- passing SQL-result-ind 235, 239
- passing SQL-state 235
- performance 201
- process of implementation 205
- referring to functions 210
- referring to table functions 211

UDFs (User-defined functions) (*continued*)

- registering 205
- registering UDFs 205
 - examples of registering 205
- reuse 201
- RETURNS TABLE clause 234, 238, 239
- runtime environment 231
- save and restore considerations 206
- scalar function
 - error processing 241
- scalar functions 204
- schema-name 203
- schema-name and UDFs 203
- signature, two functions and the same 203
- sourced 219
- summary of function references 213
- synergy with UDTs and LOBs
 - examples of complex applications 223
- table function
 - considerations 240
 - error processing 241
- table function example 209
- table functions 204
- thread considerations 232
- type of functions 204
- UDFs to query instances of UDTs
 - example 224
- unqualified reference 203
- user-defined sourced functions on UDTs 220
- why use UDFs 200
- writing function code 232
- writing your own UDF 231
 - external 234
 - SQL 233

UDTs (User-defined types)

- and DB2 object extensions 189
- assignments in dynamic SQL
 - example 221
- assignments involving different UDTs
 - example 221
- assignments involving UDTs
 - example 221
- casting between UDTs 219
- comparisons involving UDTs
 - example 218, 220
- defining a UDT 216
- defining tables 217
- defining tables with UDTs 217
- defining the UDT and UDFs
 - example 223
- definition 9
- iSeries Navigator
 - defining 284
- LOB locators to manipulate UDT
 - instances example 225
- manipulating
 - examples of 218
- resolving unqualified UDTs 216
- strong typing 218
- synergy with UDFs and LOBs
 - examples of complex applications 223

- UDTs (User-defined types) *(continued)*
 - UDFs to query instances of UDTs
 - example 224
 - use of UDTs in UNION example 222
 - user-defined sourced functions on UDTs 220
 - using CREATE DISTINCT TYPE
 - example 216
 - why use UDTs 215
- UNION ALL 88
- UNION keyword 85
 - use of UDTs in UNION example 222
- unit of work
 - boundary for package creation 332
 - distributed 327
 - effect on open cursor 133
 - package creation 332
 - remote 327
 - rollback required 345
- unprotected resource 340
- UPDATE statement 93
 - and referential constraints 140
 - changing data
 - with host variables 98
 - correlated subquery, using in 112
 - description 97
 - FOR UPDATE OF
 - restrictions 101
 - identity column
 - example 100
 - scalar subselect
 - example 99
 - SET clause
 - column name 98
 - constant 98
 - DEFAULT 98
 - expression 98
 - host variable 98
 - NULL value 98
 - scalar subselect 98
 - special register 98
 - SET CLAUSE 98
 - update rules for constraints 140
 - updating data as it is retrieved
 - example 100
 - using SELECT
 - example 99
 - WHERE clause
 - example 27
- user auxiliary storage pool (ASP) 320
- user profile
 - authorization ID 3
 - authorization name 3
- user source file member
 - definition 11
- USER special register 71
- user-defined relationship 273
- USING
 - clause
 - dynamic SQL 265
 - DESCRIPTOR clause 267

V

- view
 - creating 31, 55
 - CREATE VIEW statement 30

- view *(continued)*
 - creating over multiple tables 31
 - creating using iSeries Navigator 42, 44
 - creating with iSeries Navigator
 - with iSeries Navigator 42
 - definition 3, 7
 - deleting using iSeries Navigator 46
 - limiting access 30
 - read-only 56
 - security 306
 - sort sequence 118
 - using 55
 - WITH CASCADED CHECK 145
 - WITH CHECK 144
 - WITH LOCAL CHECK 146

W

- WHENEVER NOT FOUND clause 126
- WHERE clause
 - AND 77
 - column name 63
 - comparison operators 65
 - constant 64
 - description 63
 - example
 - dynamic SQL 267
 - expressions 63
 - host variable 64
 - joining tables 79
 - multiple search condition within
 - a 77
 - NOT 77
 - NOT keyword 65
 - NULL value 64
 - OR 77
 - special register 64
 - subquery 64
- WHERE CURRENT OF clause 127

X

- X/Open call level interface 2



Printed in U.S.A.