IBM

System i

Database
SQL programming

Version 6 Release 1

IBM

System i

Database
SQL programming

Version 6 Release 1

> **Note**
>
> Before using this information and the product it supports, read the information in "Notices," on page 299.

Contents

SQL programming

The DB2® for i5/OS® database provides a wide range of support for Structured Query Language (SQL).

The examples of SQL statements shown in this topic collection are based on the sample tables and assume that the following statements are true:

- 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.
- 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 the 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 single-quotation marks (').
- A sort sequence of *HEX is used, unless otherwise noted.

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

Because this topic collection 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.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

    Related concepts

    Embedded SQL programming

    Related reference

    "DB2 for i5/OS sample tables" on page 277
    These sample tables are referred to and used in the SQL programming and the SQL reference topic collections.

    DB2 for i5/OS SQL reference

What's new for V6R1

Read about new or significantly changed information for the SQL programming topic collection.

Row change timestamp column

When you create a table, you can now define a column to be a row change timestamp column. Every time a row is added or changed in the table, the row change timestamp column value is set to the timestamp of the insert or update operation. For more information, see "Creating a row change timestamp column" on page 22.

Row change expressions

The new expressions ROW CHANGE TIMESTAMP and ROW CHANGE TOKEN can be used to determine when a row was last changed. For more information, see "Row change expressions" on page 51.

Full outer join

DB2 for i5/OS now supports full outer joins. Previously, you could only simulate a full outer join by using a left outer join and a right exception join. For a full outer join example, see "Full outer join" on page 62.

SELECT from INSERT

When you insert one or more rows into a table, you can now select the result rows of the insert operation by specifying the INSERT statement in the FROM clause of a SELECT statement. For more information, see "Selecting inserted values" on page 85.

Decimal floating point support

DB2 for i5/OS SQL now supports the decimal floating-point data type:
- "Allowable conversions of data types" on page 29
- "Data retrieval errors" on page 79
- "Parameter passing conventions for stored procedures and user-defined functions" on page 145

Source stream file support

Either a source file member or a source stream file can now contain the source for a DB2 for i5/OS application program. The SQL statement processor now allows statements to be run from a source stream file as well. For more information, see the following topics:
- "SQL concepts" on page 4
- "Application program objects" on page 12
- "User source file" on page 14
- "Using the SQL statement processor" on page 251
- "Source listing for the SQL statement processor" on page 253

Other functional changes to the SQL programming information
- The default value, the hidden attribute, and the row change timestamp attribute can also be included in the LIKE or AS clause of the CREATE TABLE statement. See "Creating a table using LIKE" on page 20 and "Creating a table using AS" on page 20.
- Most expressions allowed by SQL can now be used in the definition of the key columns for indexes. For an example, see "Creating indexes" on page 36.
- New special registers have been added to the "Special registers in SQL statements" on page 48 topic.
- The "Concurrency" on page 110 topic has been updated to include considerations for using the SKIP LOCKED DATA clause.
- The "Indicator variables and stored procedures" on page 150 topic has been updated to document support for extended indicators.

Nontechnical changes to the SQL programming information
- Information about accessing DB2 for i5/OS data through client interfaces has been moved from the SQL programming topic collection to the Database administration topic collection.

- The example of external triggers has been moved to the Example: Trigger program topic in the Database programming topic collection.

How to see what's new or changed

To help you see where technical changes have been made, the information center uses:
- The ≫ image to mark where new or changed information begins.
- The ≪ image to mark where new or changed information ends.

In PDF files, you might see revision bars (|) in the left margin of new and changed information.

To find other information about what's new or changed this release, see the Memo to users.

PDF file for SQL programming

You can view and print a PDF file of this information.

To view or download the PDF version of this document, select SQL programming (about 3900 KB).

Saving PDF files

To save a PDF on your workstation for viewing or printing:
1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe® Reader installed on your system to view or print these PDFs. You can download a free

copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

Introduction to DB2 for i5/OS Structured Query Language

Structured Query Language (SQL) is a standardized language for defining and manipulating data in a relational database. These topics describe the System i® implementation of the SQL using the DB2 for i5/OS database and the IBM® DB2 Query Manager and SQL Development Kit for i5/OS 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. For information about embedded SQL, see Embedded SQL programming.

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.

SQL can access data in a remote relational database, using the IBM Distributed Relational Database Architecture™ (DRDA®).

> **Related concepts**
> Distributed database programming
> **Related reference**

"Distributed relational database function and SQL" on page 254

A *distributed relational database* consists of a set of SQL objects that are spread across interconnected computer systems.

SQL concepts

DB2 for i5/OS SQL consists of several main parts, such as SQL runtime support, precompilers, and interactive SQL.

- SQL runtime support

  SQL run time parses SQL statements and runs any SQL statements. This support is part of the i5/OS licensed program, which allows applications that contain SQL statements to be run on systems where the IBM DB2 Query Manager and SQL Development Kit for i5/OS 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++
  - ILE COBOL
  - COBOL
  - PL/I
  - RPG III (part of RPG)
  - ILE RPG

  The SQL host language precompilers prepare an application program that contains SQL statements. The host language compilers then compile the precompiled host source programs. For more information about precompiling, see Preparing and running a program with SQL statements in the Embedded SQL programming information. The precompiler support is part of the IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program.

- SQL interactive interface

  The SQL interactive interface allows you to create and run SQL statements. For more information about interactive SQL, see "Using interactive SQL" on page 240. Interactive SQL is part of the IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program.

- Run SQL Scripts

  The Run SQL Scripts window in System i Navigator allows you to create, edit, run, and troubleshoot scripts of SQL statements.

- Run SQL Statements (RUNSQLSTM) CL command

  The RUNSQLSTM command can be used to run a series of SQL statements that are stored in a source file or a source stream file. For more information about the RUNSQLSTM command, see "Using the SQL statement processor" on page 251.

- DB2 Query Manager

  DB2 Query Manager 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 IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program. For more information, see Query

  Manager Use .

- 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 Coding SQL statements in REXX applications in the Embedded SQL programming information.

- SQL call level interface

  The DB2 for i5/OS database supports the SQL call level interface. This allows users of any of the ILE languages to access SQL functions directly through bound calls to a service program that is provided

by the system. Using the SQL call level interface, you can perform all the SQL functions without the need to precompile. This is a standard set of procedure calls to prepare SQL statements, run SQL statements, fetch rows of data, and even perform 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 SQL call level interface in the Database section of the i5/OS Information Center.

- Process Extended Dynamic SQL (QSQPRCED) API

  This application programming interface (API) provides an extended dynamic SQL capability. You can prepare SQL statements into an SQL package and run them by using this API. Statements that are prepared into a package by this API persist until the package or statement is explicitly dropped. For more information about the QSQPRCED API, see Process Extended Dynamic SQL (QSQPRCED) API. For general information about APIs, see Application programming interfaces.

- Syntax Check SQL Statement (QSQCHKS) API

  This API syntax checks SQL statements. For more information about the QSQCHKS API, see Syntax Check SQL Statement (QSQCHKS) API. For general information about APIs, see Application programming interfaces.

- DB2 Multisystem

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

- DB2 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 toward 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. For more information, see Controlling parallel processing for queries in the Database performance and query optimization topic collection.

SQL relational database and system terminology

In the relational model of data, all data is perceived as existing in tables. DB2 for i5/OS objects are created and maintained as system objects.

The following table shows the relationship between system terms and SQL relational database terms.

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.**

Related concepts

Distributed database programming

SQL and system naming conventions

You can use either the system (*SYS) or the SQL (*SQL) naming convention in DB2 for i5/OS
programming.

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 by using the SET OPTION statement.

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 schema
name in the form:

```
schema.table
```

> **Related reference**
>
> Qualification of unqualified object names

Types of SQL statements

There are several basic types of SQL statements. They are listed here according to their functions.

- SQL schema statements, also known as data definition language (DDL) statements
- SQL data and data change statements, also known as data manipulation language (DML) statements
- Dynamic SQL statements
- Embedded SQL host language statements

SQL schema statements
| ALTER FUNCTION
| ALTER PROCEDURE
ALTER SEQUENCE
ALTER TABLE
COMMENT ON
CREATE ALIAS
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE INDEX
CREATE PROCEDURE
CREATE SCHEMA
CREATE SEQUENCE
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DROP ALIAS
DROP DISTINCT TYPE
DROP FUNCTION
DROP INDEX
DROP PACKAGE
DROP PROCEDURE
DROP SEQUENCE
DROP SCHEMA
DROP TABLE
DROP TRIGGER
DROP VIEW
GRANT DISTINCT TYPE
GRANT FUNCTION
GRANT PACKAGE
GRANT PROCEDURE
GRANT SEQUENCE
GRANT TABLE
LABEL ON
RENAME
REVOKE DISTINCT TYPE
REVOKE FUNCTION
REVOKE PACKAGE
REVOKE PROCEDURE
REVOKE SEQUENCE
REVOKE TABLE

SQL data statements
CLOSE
DECLARE CURSOR
DELETE
FETCH
FREE LOCATOR
HOLD LOCATOR
INSERT
LOCK TABLE
OPEN
REFRESH TABLE
SELECT INTO
SET variable
UPDATE
VALUES INTO

SQL data change statements
DELETE
INSERT
UPDATE

SQL connection statements
CONNECT
DISCONNECT
RELEASE
SET CONNECTION

SQL transaction statements
COMMIT
| RELEASE SAVEPOINT
ROLLBACK
SAVEPOINT
SET TRANSACTION

SQL session statements
DECLARE GLOBAL TEMPORARY TABLE
SET CURRENT DECFLOAT ROUNDING MODE
SET CURRENT DEGREE
SET ENCRYPTION PASSWORD
SET PATH
SET SCHEMA
SET SESSION AUTHORIZATION

Dynamic SQL statements	Embedded SQL host language statements
ALLOCATE DESCRIPTOR	BEGIN DECLARE SECTION
DEALLOCATE DESCRIPTOR	DECLARE PROCEDURE
DESCRIBE	DECLARE STATEMENT
DESCRIBE INPUT	DECLARE VARIABLE
DESCRIBE TABLE	END DECLARE SECTION
EXECUTE	GET DIAGNOSTICS
EXECUTE IMMEDIATE	INCLUDE
GET DESCRIPTOR	SET OPTION
PREPARE	SET RESULT SETS
SET DESCRIPTOR	SIGNAL
	WHENEVER

SQL control statements
CALL

SQL statements can operate on objects that are created by SQL as well as externally described physical files and single-format logical files. They do not refer to the interactive data definition utility (IDDU) dictionary definition for program-described files. Program-described files appear as a table with only a single column.

Related concepts

"Data definition language" on page 15
Data definition language (DDL) describes the portion of SQL that creates, alters, and deletes database objects. These database objects include schemas, tables, views, sequences, catalogs, indexes, and aliases.

"Data manipulation language" on page 38
Data manipulation language (DML) describes the portion of SQL that manipulates or controls data.

Related reference

DB2 for i5/OS SQL reference

SQL communication area

The SQL communication area (SQLCA) is a set of variables that provides an application program with information about its execution of SQL statements. The SQLCA is updated at the end of the execution of every SQL statement.

Related concepts

SQLCA (SQL communication area)

Handling SQL error return codes using the SQLCA

SQL diagnostics area

The SQL diagnostics area maintained by the database manager provides information about the SQL statement that is most recently run. Your application program can access the SQL diagnostics area using the GET DIAGNOSTICS statement.

Related concepts

Using the SQL diagnostics area

Related reference

GET DIAGNOSTICS statement

SQL objects

SQL *objects* are schemas, journals, catalogs, tables, aliases, views, indexes, constraints, triggers, sequences, stored procedures, user-defined functions, user-defined types, and SQL packages. SQL creates and maintains these objects as system objects.

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.

Journals and journal receivers

A *journal* and a *journal receiver* are used to record changes to tables and views in the database.

Journals and journal receivers are used in processing the SQL COMMIT, ROLLBACK, SAVEPOINT, and RELEASE SAVEPOINT statements. Journals and journal receivers can also be used as audit trails or for forward or backward recovery.

> **Related concepts**
> Journal management
> Commitment control

Catalogs

An SQL *catalog* is a collection of tables and views that describe tables, views, indexes, packages, procedures, functions, files, sequences, 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.

> **Related reference**
> Catalog

Tables, rows, and columns

A *table* is a two-dimensional arrangement of data that consists 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 non-keyed physical file.

A *materialized query table* is a table that is used to contain materialized data that is derived from one or more source tables specified by a select-statement.

A *partitioned table* is a table whose data is contained in one or more local partitions (members).

> **Related concepts**
> DB2 Multisystem
> **Related reference**
> Data types

"Creating and altering a materialized query table" on page 21

A *materialized query table* is a table whose definition is based on the result of a query, and whose data is in the form of precomputed results that are taken from the table or tables on which the materialized query table definition is based.

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.

Related reference

Aliases

Views

A *view* appears like a table to an application program. However, a view contains no data and only logically represents one or more tables over which it is created.

A view can contain all the columns and rows of the given tables or a 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.

Related reference

Views

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 the applications that change the table.

Related concepts

Creating an index strategy

Constraints

A *constraint* is a rule enforced by the database manager to limit the values that can be inserted, deleted, or updated in a table.

DB2 for i5/OS 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. You can create a unique constraint using the CREATE TABLE or ALTER TABLE statement. Although the CREATE INDEX statement 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 the 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 one of the following conditions is met:

- They appear as values of a parent key.
- 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 the rule that limits the values allowed in a column or group of columns. You can create a check constraint using the CREATE TABLE or ALTER TABLE statement. 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 (because of a null value).

    Related reference

    "Constraints" on page 118
    The DB2 for i5/OS database supports unique, referential, and check constraints.

Triggers

A *trigger* is a set of actions that runs automatically whenever a specified event occurs to a specified table or view.

An event can be an insert, an update, a delete, or a read operation. A trigger can run either before or after the event. DB2 for i5/OS supports SQL insert, update, and delete triggers and external triggers.

Related tasks

Triggering automatic events in your database

Stored procedures

A *stored procedure* is a program that can be called with the SQL CALL statement.

DB2 for i5/OS supports external procedures and SQL procedures. An external procedure can be any system program, service program, or REXX procedure. It cannot be a System/36™ program or procedure. An SQL procedure is defined entirely in SQL and can contain SQL statements, including SQL control statements.

Related concepts

"Stored procedures" on page 122
A *procedure* (often called a stored procedure) is a program that can be called to perform operations. A procedure can include both host language statements and SQL statements. Procedures in SQL provide the same benefits as procedures in a host language.

Sequences

A *sequence* is a data area object that provides a quick and easy way of generating unique numbers.

You can use a sequence to replace an identity column or a user-generated numeric column. A sequence has uses similar to these alternatives.

Related reference

"Creating and using sequences" on page 24
Sequences are similar to identity columns in that they both generate unique values. However, sequences are objects that are independent of any tables. You can use sequences to generate values quickly and easily.

User-defined functions

A *user-defined function* is a program that can be called like any built-in functions.

DB2 for i5/OS supports external functions, SQL functions, and sourced functions. An external function 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 function or an external function.

Related concepts

"Using user-defined functions" on page 152
In writing SQL applications, you can implement some actions or operations as a user-defined function (UDF) or as a subroutine in your application. Although it might appear easier to implement new operations as subroutines, you might want to consider the advantages of using a UDF instead.

User-defined types

A *user-defined type* is a distinct data type that you can define independently of the data types that are provided by the database management system.

Distinct data types map on a one-to-one basis to existing database types.

Related concepts

"User-defined distinct types" on page 199
A user-defined distinct type (UDT) is a mechanism to extend DB2 capabilities beyond the built-in data types that are available.

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 (CRTSQL*xxx*) command and a program object is created. Packages can also be created with the Create SQL Package (CRTSQLPKG) command.

Note: The *xxx* in this command refers to the host language indicators: CI for ILE C, CPPI for ILE C++, CBL for COBOL, CBLI for ILE COBOL, PLI for PL/I, RPG for RPG/400®, and RPGI for ILE RPG.

SQL packages can also be created with the Process Extended Dynamic SQL (QSQPRCED) API. The SQL packages mentioned within this topic collection refer exclusively to distributed program SQL packages. The QSQPRCED API uses SQL packages to provide extended dynamic SQL support.

Related reference

"Distributed relational database function and SQL" on page 254
A *distributed relational database* consists of a set of SQL objects that are spread across interconnected computer systems.
Process Extended Dynamic SQL (QSQPRCED) API

Application program objects

Several objects are created when a DB2 for i5/OS application program is being precompiled.

DB2 for i5/OS supports both non-ILE and ILE precompilers. Application programs can be either distributed or nondistributed.

With the DB2 for i5/OS database, you might 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 for i5/OS program, you must manage only the original source and
| the resulting program. The following figure shows the objects involved and the steps that happen during
| the precompile and compile processes for a nondistributed non-ILE DB2 for i5/OS program. The user
| source file precompiles the source to a temporary source file member. This member is then compiled into
| a program.

```
+-----------+              +------------+             +-----------+
| User      | Precompile   | Temporary  | Compile     | Program   |
| Source    |------------->| Source     |------------>|           |
| File      |              | File       |             |           |
+-----------+              | Member     |             |-----------|
                           +------------+             |           |
                           | Processed  |             | Access    |
                           | SQL        |             | Plan      |
                           | Statements |             |           |
                           +------------+             +-----------+
                                                         RV2W565-2
```
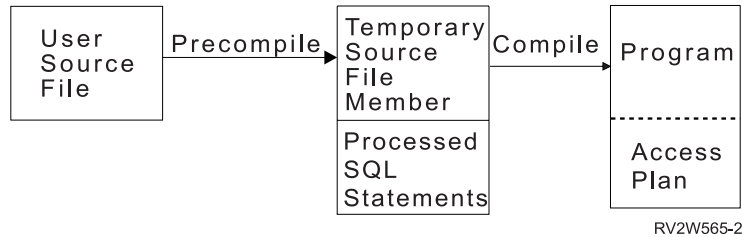
| With a nondistributed ILE DB2 for i5/OS program, you might need to manage the original source, the
| modules, and the resulting program or service program. The following figure shows the objects involved
| and the steps that happen during the precompile and compile processes for a nondistributed ILE DB2 for
| i5/OS program when OBJTYPE(*PGM) is specified on the precompile command. The user source file
| precompiles the source to a temporary source file member. This member is then compiled into a module
| that binds to a program.

```
+-----------+            +------------+            +-----------+         +-----------+
| User      | Precompile | Temporary  | Compile    | Module    | Bind    | Program   |
| Source    |----------->| Source     |----------->|           |-------->|           |
| File      |            | File       |            |           |         |-----------|
+-----------+            | Member     |            |-----------|         |           |
                         +------------+            | Processed |         | Access    |
                         | Processed  |            | SQL       |         | Plan      |
                         | SQL        |            | Statements|         |           |
                         | Statements |            +-----------+         +-----------+
                         +------------+                                   RV2W569-1
```
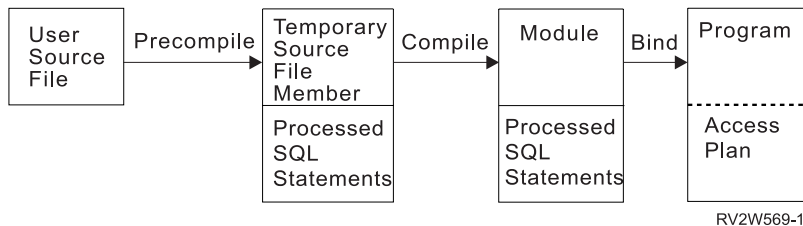
| With a distributed non-ILE DB2 for i5/OS program, you must manage the original source, the resulting
| program, and the resulting package. The following figure shows the objects and the steps that occur
| during the precompile and compile processes for a distributed non-ILE DB2 for i5/OS program. The user
| source file precompiles the source to a temporary source file member. This member is then compiled into
| a program. After the program is created, an SQL package is created to hold the program.

```
+-----------+          +------------+          +-----------+  Create   +-----------+
| User      |Precompile| Temporary  | Compile  | Program   |  SQL      | SQL       |
| Source    |--------->| Source     |--------->|           |---------->| Package   |
| File      |          | File       |          |-----------|  Package  |-----------|
+-----------+          | Member     |          |           |           |           |
                       +------------+          | Access    |           | Access    |
                       | Processed  |          | Plan      |           | Plan      |
                       | SQL        |          |           |           |           |
                       | Statements |          +-----------+           +-----------+
                       +------------+                                    RV2W566-3
```
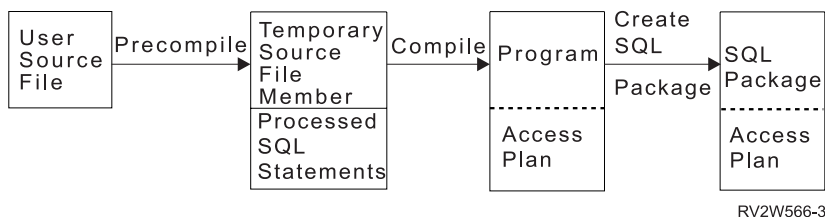
| With a distributed ILE DB2 for i5/OS program, you must manage the original source, module objects, the
| resulting program or service program, and the resulting packages. An SQL package can be created for
| each distributed module in a distributed ILE program or service program. The following figure shows the
| objects and the steps that occur during the precompile and compile processes for a distributed ILE DB2
| for i5/OS program. The user source file precompiles the source to a temporary source file member. This
| member is then compiled into a module that binds to a program. After the program is created, an SQL
| package is created to hold the program.

RV2W570-2
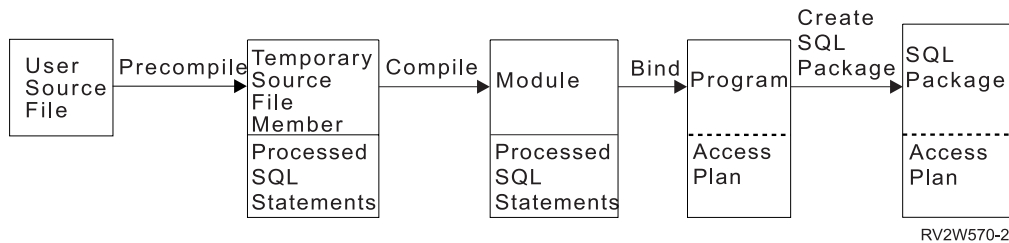
Note: The access plans associated with the DB2 for i5/OS distributed program object are not created until the program is run locally.

> **Related tasks**
>
> Preparing and running a program with SQL statements

User source file

A source file member or a source stream file contains the 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 System i licensed program.

Output source file member

By default, the precompile process creates a temporary source file QSQLT*xxxxx* in the QTEMP library. However, you can specify the output source file as a permanent file on the precompile command.

If the precompile process uses the QTEMP library, the system automatically deletes the file when the job is completed. 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 runtime support, which have replaced embedded SQL statements
* Parsed and syntax-checked SQL statements

By default, the precompiler calls the host language compiler.

> **Related tasks**
>
> Preparing and running a program with SQL statements

Program

A *program* is an object that is created as a result of the compilation process for non-ILE compilations or as a result of the bind process for ILE compilations.

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 been successfully created. Access plans are not created during program creation for SQL statements if the statements refer to an object, such as a table or view, that cannot be found or 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 non-distributed 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:
* You successfully create a distributed SQL program by specifying the relational database (RDB) parameter on the CREATE SQL (CRTSQL*xxx*) commands.

- You run the Create SQL Package (CRTSQLPKG) command.

When a distributed SQL program is created, the name of the SQL package and an internal consistency token are saved in the program. They 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 you create by compiling source code using the Create Module (CRT*xxx*MOD) command (or any of the Create Bound Program (CRTBND*xxx*) 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 typically 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.

Related reference

Create Program (CRTPGM) command

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.

Data definition language

Data definition language (DDL) describes the portion of SQL that creates, alters, and deletes database objects. These database objects include schemas, tables, views, sequences, catalogs, indexes, and aliases.

Related concepts

"Types of SQL statements" on page 6
There are several basic types of SQL statements. They are listed here according to their functions.

Related tasks

Getting started with SQL

Creating a schema

A schema provides a logical grouping of SQL objects. To create a schema, use the CREATE SCHEMA statement.

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 libraries. 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.

You can create a schema using the CREATE SCHEMA statement. For example, create a schema called DBTEMP:

CREATE SCHEMA DBTEMP

> **Related reference**
>
> CREATE SCHEMA

Creating a table

A table can be visualized as a two-dimensional arrangement of data that consists of rows and columns. To create a table, use the CREATE TABLE statement.

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 non-keyed physical file.

You can create a table using the CREATE TABLE statement. The definition must include its name and the names and attributes of its columns. The definition can include other attributes of the table, such as the primary key.

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

- Part number: Integer between 1 and 9999, and 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))
```

> **Related concepts**
>
> Data types

Adding and removing constraints

Constraints can be added to a new table or to an existing table. To add a unique or primary key, a referential constraint, or a check constraint, use the CREATE TABLE or the ALTER TABLE statement. To remove a constraint, use the ALTER TABLE statement.

For example, add 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, replace the keyword PRIMARY with UNIQUE.

You can remove a constraint using the same ALTER TABLE statement:

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

Referential integrity and tables

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:
- 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 that 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 that 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, 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 that states that every non-null foreign key must have a matching parent key.

SQL supports the referential integrity concept with the CREATE TABLE and ALTER TABLE statements.

> **Related reference**
> "DB2 for i5/OS sample tables" on page 277
> These sample tables are referred to and used in the SQL programming and the SQL reference topic collections.
> CREATE TABLE
> ALTER TABLE

Adding and removing referential constraints:

You can use the CREATE TABLE statement or the ALTER TABLE statement to add a referential constraint. To remove a referential constraint, use the ALTER TABLE statement.

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.

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.

After 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 System i Navigator, CL commands, utilities, or high-level language statements.

> **Related reference**
> CREATE TABLE
> ALTER TABLE

Example: Adding referential constraints:

You define a referential constraint that every department number in the sample employee table must appear in the department table. The referential 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,
       FIRSTNME  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 LASTNAME 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.

Example: Removing constraints

When you remove the primary key over the DEPTNO column in the DEPARTMENT table, other tables are affected.

You also remove the REPORTS_TO_EXISTS constraint that is defined on the DEPARTMENT table and the WORKDEPT_EXISTS constraint that is defined on the EMPLOYEE table, because the primary key that you remove 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
```

Check pending

Referential constraints and check constraints can be in a check pending state, where potential violations of the constraints 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 that are limited by the check constraint. When the system determines that a constraint might 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.
- Insert and update operations are allowed and the constraint is enforced.

To get a constraint out of check pending, follow these steps:

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.

> **Related concepts**
>
> Check pending status in referential constraints

Working with constraints that are in check pending status

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 following definitions are copied:
* Column names (and system column names)
* Data type, length, precision, and scale
* CCSID

If the LIKE clause immediately follows the table name and is not enclosed in parentheses, the following attributes are also included:
| • Column text (LABEL ON)
| • Column heading (LABEL ON)
* Default value
| • Hidden attribute
| • Identity attribute
* Nullability
| • Row change timestamp attribute

| If the specified table or view contains an identity column, you must specify the option 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. There are similar options to include
| the default value, the hidden attribute, and the row change timestamp attribute. 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
```
   Related reference
   CREATE TABLE

Creating a table using AS

You can create a table from the result of a SELECT statement. To create this type of table, use the CREATE TABLE AS 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 the option 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. There are similar options to include
| the default value, the hidden attribute, and the row change timestamp attribute. The WITH NO DATA
| clause indicates that the column definitions are to be copied without the data. If you want to include the

| data in the new table EMPLOYEE3, include the WITH DATA clause. If the specified query includes a
| non-SQL-created physical file or logical file, any non-SQL result attributes are removed.

Related concepts

"Retrieving data using the SELECT statement" on page 38
The SELECT statement tailors your query to gather data. You can use the SELECT statement to
retrieve a specific row or retrieve data in a specific way.

Related reference

CREATE TABLE

Creating and altering a materialized query table

A *materialized query table* is a table whose definition is based on the result of a query, and whose data is in
the form of precomputed results that are taken from the table or tables on which the materialized query
table definition is based.

If the optimizer determines that a query runs faster against a materialized query table than it does
against the base table or tables, the query will run against the materialized query table. You can directly
query a materialized query table. For more information about how the optimizer uses materialized query
tables, see the Database performance and query optimization topic.

Assume a very large transaction table named TRANS contains one row for each transaction processed by
a company. The table is defined with many columns. Create a materialized query table for the TRANS
table that contains daily summary data for the date and amount of a transaction by issuing the following:

```
CREATE TABLE STRANS
  AS (SELECT YEAR AS SYEAR, MONTH AS SMONTH, DAY AS SDAY, SUM(AMOUNT) AS SSUM
      FROM TRANS
      GROUP BY YEAR, MONTH, DAY )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER
```

This materialized query table specifies that the table is not populated at the time that it is created by
using the DATA INITIALLY DEFERRED clause. REFRESH DEFERRED indicates that changes made to
TRANS are not reflected in STRANS. Additionally, this table is maintained by the user, enabling the user
to use ALTER, INSERT, DELETE, and UPDATE statements.

To populate the materialized query table or refresh the table after it has been populated, use the
REFRESH TABLE statement. This causes the query associated with the materialized query table to be run
and causes the table to be filled with the results of the query. To populate the STRANS table, run the
following statement:

```
REFRESH TABLE STRANS
```

You can create a materialized query table from an existing base table as long as the result of the
select-statement provides a set of columns that match the columns in the existing table (same number of
columns and compatible column definitions). For example, create a table TRANSCOUNT. Then, change
the base table TRANSCOUNT into a materialized query table:

To create the table:

```
CREATE TABLE TRANSCOUNT
    (ACCTID SMALLINT NOT NULL,
     LOCID SMALLINT,
     YEAR DATE
     CNT INTEGER)
```

You can alter this table to be a materialized query table:

```
ALTER TABLE TRANSCOUNT
  ADD MATERIALIZED QUERY
     (SELECT  ACCTID, LOCID, YEAR, COUNT(*) AS CNT
      FROM TRANS
      GROUP BY ACCTID, LOCID, YEAR )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER
```

Finally, you can change a materialized query table back to a base table. For example:

```
ALTER TABLE TRANSCOUNT
     DROP MATERIALIZED QUERY
```

In this example, the table TRANSCOUNT is not dropped, but it is no longer a materialized query table.

Related concepts

"Tables, rows, and columns" on page 9

A *table* is a two-dimensional arrangement of data that consists of *rows* and *columns*.

Declaring a global temporary table

You can create a temporary table for use with your current session. To create a temporary table, use 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 end your session, the rows of the table are deleted and the table is dropped.

The syntax of this statement is similar to that of the CREATE TABLE statement and can include the LIKE or 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, the same as any other table. You can drop this table by issuing the DROP TABLE statement:

```
DROP TABLE ORDERS
```

Related reference

DECLARE GLOBAL TEMPORARY TABLE

Creating a row change timestamp column

Every time a row is added or changed in a table with a row change timestamp column, the row change timestamp column value is set to the timestamp corresponding to the time of the insert or update operation.

The data type of a row change timestamp column must be TIMESTAMP. You can define only one row change timestamp column in a table.

When you create a table, you can define a column in the table to be a row change timestamp column. For example, create a table ORDERS with columns called ORDERNO, SHIPPED_TO, ORDER_DATE, STATUS, and CHANGE_TS. Define CHANGE_TS as a row change timestamp column.

```
CREATE TABLE ORDERS
   (ORDERNO SMALLINT,
    SHIPPED_TO VARCHAR(36),
```

```
|       ORDER_DATE DATE,
|       STATUS CHAR(1),
|       CHANGE_TS TIMESTAMP FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP)
```

When a row is inserted into the ORDERS table, the CHANGE_TS column for the row is set to the
timestamp of the insert operation. Any time a row in ORDERS is updated, the CHANGE_TS column for
the row is modified to reflect the timestamp of the update operation.

You can drop the row change timestamp attribute from a column:

```
ALTER TABLE ORDER
  ALTER COLUMN CHANGE_TS
  DROP ROW CHANGE TIMESTAMP
```

The column CHANGE_TS remains as a TIMESTAMP column in the table, but the system no longer
automatically updates timestamp values for this column.

Creating and altering an identity column

Every time a row is added to a table with an identity column, the identity column value for the new row
is generated 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 three 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 a 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 32 767. When this column value
reaches 32 767, it will restart at 500 again. If 500 is still assigned to a column, and a unique key is
specified on the identity column, a duplicate key error is returned. The next insert operation 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, specify the column to be data type INTEGER or even BIGINT. If you want
the value of the identity column to decrease, 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, you want 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.

> **Related reference**
>
> "Comparison of identity columns and sequences" on page 26
> While identity columns and sequences are similar in many ways, there are also differences.
>
> "Inserting values into an identity column" on page 84
> You can insert a value into an identity column or allow the system to insert a value for you.
>
> "Updating an identity column" on page 89
> You can update the value in an identity column to a specified value or have the system generate a new value.

Using ROWID

Using ROWID is another way to have the system assign a unique value to a column. 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 and using sequences

Sequences are similar to identity columns in that they both generate unique values. However, sequences are objects that are independent of any tables. You can use sequences to generate values quickly and easily.

Sequences are not tied to a column in a table and are accessed separately. Additionally, they are not treated as any part of a transaction's unit of work.

You create a sequence using the CREATE SEQUENCE statement. For an example similar to the identity column example, create a sequence ORDER_SEQ:

```
CREATE SEQUENCE ORDER_SEQ
START WITH 500
INCREMENT BY 1
MAXVALUE 1000
CYCLE
CACHE 24
```

This sequence is defined with a starting value of 500, incremented by 1 for every use, and recycles when the maximum value is reached. In this example, the maximum value for the sequence is 1000. When this value reaches 1000, it will restart at 500 again.

After this sequence is created, you can insert values into a column using the sequence. For example, insert the next value of the sequence ORDER_SEQ into a table ORDERS with columns ORDERNO and CUSTNO.

First, create the table ORDERS:

```
CREATE TABLE ORDERS
(ORDERNO SMALLINT NOT NULL,
CUSTNO SMALLINT);
```

Then, insert the sequence value:
```
INSERT INTO ORDERS (ORDERNO, CUSTNO)
VALUES (NEXT VALUE FOR ORDER_SEQ, 12)
```

Running the following statement returns the values in the columns:
```
SELECT *
FROM ORDERS
```

Table 2. Results for SELECT from table ORDERS

ORDERNO	CUSTNO
500	12

In this example, the next value for sequence ORDER is inserted into the ORDERNO column. Issue the INSERT statement again. Then run the SELECT statement.

Table 3. Results for SELECT from table ORDERS

ORDERNO	CUSTNO
500	12
501	12

You can also insert the previous value for the sequence ORDER by using the PREVIOUS VALUE expression. You can use NEXT VALUE and PREVIOUS VALUE in the following expressions:

- Within the *select-clause* of a SELECT statement or SELECT INTO statement as long as the statement does not contain a DISTINCT keyword, a GROUP BY clause, an ORDER BY clause, a UNION keyword, an INTERSECT keyword, or an EXCEPT keyword
- Within a VALUES clause of an INSERT statement
- Within the *select-clause* of the fullselect of an INSERT statement
- Within the SET clause of a searched or positioned UPDATE statement, though NEXT VALUE cannot be specified in the *select-clause* of the subselect of an expression in the SET clause

You can alter a sequence by issuing the ALTER SEQUENCE statement. Sequences can be altered in the following ways:

- Restarting the sequence
- Changing the increment between future sequence values
- Setting or eliminating the minimum or maximum values
- Changing the number of cached sequence numbers
- Changing the attribute that determines whether the sequence can cycle or not
- Changing whether sequence numbers must be generated in order of request

For example, change the increment of values of sequence ORDER from 1 to 5:
```
ALTER SEQUENCE ORDER_SEQ
INCREMENT BY 5
```

After this change is complete, run the INSERT statement again and then the SELECT statement. Now the table contains the following columns.

Table 4. Results for SELECT from table ORDERS

ORDERNO	CUSTNO
500	12
501	12

ORDERNO	CUSTNO
528	12

Notice that the next value that the sequence uses is a 528. At first glance, this number appears to be incorrect. However, look at the events that lead up to this assignment. First, when the sequence was originally create, a cache value of 24 was assigned. The system assigns the first 24 values for this cache. Next, the sequence was altered. When the ALTER SEQUENCE statement is issued, the system drops the assigned values and starts up again with the next available value; in this case the original 24 that was cached, plus the next increment, 5. If the original CREATE SEQUENCE statement did not have the CACHE clause, the system automatically assigns a default cache value of 20. If that sequence was altered, then the next available value is 25.

Related concepts

"Sequences" on page 11
A *sequence* is a data area object that provides a quick and easy way of generating unique numbers.

Comparison of identity columns and sequences

While identity columns and sequences are similar in many ways, there are also differences.

Examine these differences before you decide which to use.

An identity column has the following characteristics:

- An identity column can be defined as part of a table only when the table is created. After a table is created, you cannot alter it to add an identity column. (However, existing identity column characteristics can be altered.)
- An identity column automatically generates values for a single table.
- When an identity column is defined as GENERATED ALWAYS, the values used are always generated by the database manager. Applications are not allowed to provide their own values when changing the contents of the table.
- The IDENTITY_VAL_LOCAL function can be used to see the most recently assigned value for an identity column.

A sequence has the following characteristics:

- A sequence is a system object of type *DTAARA that is not tied to a table.
- A sequence generates sequential values that can be used in any SQL statement.
- There are two expressions used to retrieve the next values in the sequence and to look at the previous value assigned for the sequence. The PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous statement within the current session. The NEXT VALUE expression returns the next value for the specified sequence. The use of these expressions allows the same value to be used across several SQL statements within several tables.

While these are not all of the characteristics of identity columns and sequences, these characteristics can help you determine which to use depending on your database design and the applications that use the database.

Related reference

"Creating and altering an identity column" on page 23
Every time a row is added to a table with an identity column, the identity column value for the new row is generated by the system.

Creating descriptive labels using the LABEL ON statement

Sometimes a text description is useful for an object (such as a table or an index) or useful as column text or column headings. You can create a more descriptive label for these names by using the LABEL ON statement.

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 an object or a column cannot be more than 50 bytes and the label for a column heading cannot be more than 60 bytes (blanks included). Here are the 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 three 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 ''
```

This LABEL ON statement provides a DBCS column heading with two levels specified:

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

This LABEL ON statement provides the 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'
```

Related reference

LABEL

Describing an SQL object using COMMENT ON

After you create an SQL object, such as a table or view, you can provide object information for future reference using the COMMENT ON statement.

The information can be 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. 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. Usually, your comment must not be

more than 2000 characters, but for sequences the maximum length is 500 characters. If the object already contains a comment, the old comment is replaced by the new one.

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 you run 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. The following example gets the comments that are added by the COMMENT ON statement in the previous example:

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

Related reference

COMMENT

Changing a table definition

You change the definition of a table by adding a column, changing an existing column definition, such as its length or default value, dropping an existing column, adding a constraint, or removing a constraint.

To change a table definition, use 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.

Related reference

ALTER TABLE

Adding a column

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.

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

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.

Related reference

ALTER TABLE

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. You can always change a character, graphic, or binary column from fixed length to varying length or LOB; or from varying length or LOB to fixed length.

When you convert to a data type with a longer length, data is padded with the appropriate pad character. When you convert to a data type with a shorter length, data might be lost because of 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.

Related reference

"Allowable conversions of data types"
When you change the data type of an existing column, the old and new attributes must be compatible.

Related information

ALTER TABLE

Allowable conversions of data types

When you change the data type of an existing column, the old and new attributes must be compatible.

Table 5. Allowable conversions

From data type	To data type
Decimal	Numeric
Decimal	Bigint, Integer, Smallint
Decimal	Decfloat
Decimal	Float
Numeric	Decimal
Numeric	Bigint, Integer, Smallint
Numeric	Decfloat
Numeric	Float
Bigint, Integer, Smallint	Decimal
Bigint, Integer, Smallint	Numeric
Bigint, Integer, Smallint	Decfloat
Bigint, Integer, Smallint	Float
Float	Decimal
Float	Numeric
Float	Bigint, Integer, Smallint
Float	Decfloat
Character	DBCS-open
Character	UCS-2 or UTF-16 graphic
DBCS-open	Character
DBCS-open	UCS-2 or UTF-16 graphic
DBCS-either	Character
DBCS-either	DBCS-open
DBCS-either	UCS-2 or UTF-16 graphic
DBCS-only	DBCS-open
DBCS-only	DBCS graphic
DBCS-only	UCS-2 or UTF-16 graphic

Table 5. Allowable conversions (continued)

From data type	To data type
DBCS graphic	UCS-2 or UTF-16 graphic
UCS-2 or UTF-16 graphic	Character
UCS-2 or UTF-16 graphic	DBCS-open
UCS-2 or UTF-16 graphic	DBCS graphic
distinct type	*source type*
source type	*distinct type*

When you change an existing column, only the attributes that you specify are changed. All other attributes remain unchanged. For example, you have a table with 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 you run the following ALTER TABLE statement, COL2 still has an allocated length of 10 and CCSID 937, and COL3 still has an allocated length of 10.

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

Related reference

"Changing a column" on page 28
You can change a column definition in a table using the ALTER COLUMN clause of the ALTER TABLE statement.

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
```

Related reference

ALTER TABLE

Order of operations for the ALTER TABLE statement

Operations for the ALTER TABLE statement are performed in a defined order.

The ALTER TABLE statement is performed as this set of steps shows:

1. Drop constraints
2. Drop materialized query table
3. Drop partition information
4. Drop columns for which the RESTRICT option was specified
5. Alter column definitions (this includes adding columns and dropping columns for which the CASCADE option was specified)
6. Add or alter materialized query table
7. Add partitioning to a table
8. 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. To create an alias, use the CREATE ALIAS statement.

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 a table name is used. 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 MYLIB.MYALIAS is an alias for table MYLIB.MYTABLE. The following DROP statement drops 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
```

> **Related reference**
> CREATE ALIAS

Creating and using views

A view can be used to access data in one or more tables or views. You create a view by using a SELECT statement.

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

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

After you create the view, you can use it in SQL statements just like a table. You can also change the data in the base table through the view. The following SELECT statement displays the contents of EMP_MANAGERS:

```
SELECT *
     FROM CORPDATA.EMP_MANAGERS
```

The results follow.

LASTNAME	WORKDEPT
THOMPSON	B01

LASTNAME	WORKDEPT
KWAN	C01
GEYER	E01
STERN	D11
PULASKI	D21
HENDERSON	E11
SPENSER	E21

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, YEAR (CURRENT DATE - HIREDATE)
    FROM CORPDATA.EMPLOYEE
```

Because the results of querying this view change as the current year changes, they are not included here.

You can also define the previous view 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
```

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.EMPPROJACT
    WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111')
```

This view has the following data.

Table 6. Results of creating a view as UNION

EMPNO
000060
000150
000160
000170
000180
000190
000200
000210
000220
000230

Table 6. Results of creating a view as UNION (continued)

EMPNO
000240
200170
200220

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, or UCS-2 or UTF-16 graphic comparisons in the CREATE VIEW statement subselect.

You can also create views using the WITH CHECK OPTION clause to specify the level of checking when data is inserted or updated through the view.

> **Related concepts**
>
> "Retrieving data using the SELECT statement" on page 38
> The SELECT statement tailors your query to gather data. You can use the SELECT statement to retrieve a specific row or retrieve data in a specific way.
>
> "Sort sequences and normalization in SQL" on page 102
> A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.
>
> **Related reference**
>
> "Using the UNION keyword to combine subselects" on page 71
> Using the UNION keyword, you can combine two or more subselects to form a fullselect.
>
> CREATE VIEW

WITH CHECK OPTION on a view

WITH CHECK OPTION is an optional clause on the CREATE VIEW statement. It specifies the level of checking when data is inserted or updated through a view.

If WITH CHECK OPTION is specified, every row that is inserted or updated through the view must conform to the definition of the view. The 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 that do not conform to the definition of the view might be inserted or updated through the view. This means that the rows cannot be selected again through the view.

> **Related reference**
>
> CREATE VIEW

WITH CASCADED CHECK OPTION:

The WITH CASCADED CHECK OPTION clause specifies that every row that is inserted or updated through a 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 through the view.

For example, consider the following updatable 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 clause:

```
CREATE VIEW V2 AS SELECT COL1
            FROM V1 WITH CASCADED CHECK OPTION
```

The following INSERT statement fails because it produces 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:

The WITH LOCAL CHECK OPTION clause is identical to the WITH CASCADED CHECK OPTION clause except that you can update a row so that it can no longer be retrieved through the view. This can happen only 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 updatable 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 statement that failed in the previous CASCADED CHECK OPTION example succeeds 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)
```

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:

This example shows how the check option is enforced on a number of dependent views that are defined with or without a 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 statement.

- 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 need 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 need 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 because 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 means 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'.

Creating 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 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 also create an index that does not exactly match the data for a column in a table. For example, you can create an index that uses the uppercase version of an employee name:

```
CREATE INDEX CORPDATA.INX2 ON CORPDATA.EMPLOYEE (UPPER(LASTNAME))
```

Most expressions allowed by SQL can be used in the definition of the key columns.

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. One type of index, the encoded vector index (EVI), allows for faster scans that can be more easily processed in parallel.

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 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, or UCS-2 or UTF-16 graphic fields of the index.

> **Related concepts**
>
> "Sort sequences and normalization in SQL" on page 102
> A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.
>
> Creating an index strategy
>
> **Related reference**
>
> CREATE INDEX

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.

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.

> **Related reference**
>
> DB2 for i5/OS catalog views

Getting catalog information about a table

The SYSTABLES view contains a row for each table and view in the SQL schema. The SYSTABLES view provides information such as the object type (table or view), the object name, the owner of the object, and the schema where the object exists.

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

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

Getting catalog information about a column

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

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

```
SELECT *
  FROM CORPDATA.SYSCOLUMNS
  WHERE TABLE_NAME = 'DEPARTMENT'
```

The result of the previous example statement is a row of information for each column in the table.

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

```
SELECT COLUMN_NAME, TABLE_NAME, DATA_TYPE, LENGTH, HAS_DEFAULT
  FROM CORPDATA.SYSCOLUMNS
  WHERE TABLE_NAME = '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.

COLUMN_NAME	TABLE_NAME	DATA_TYPE	LENGTH	HAS_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 might also be deleted or might prevent the drop from happening.

For example, if you drop a table, any aliases, constraints, triggers, views, or indexes associated with that table are also 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
```

> **Related reference**
> DROP

Data manipulation language

Data manipulation language (DML) describes the portion of SQL that manipulates or controls data.

Related concepts

"Types of SQL statements" on page 6
There are several basic types of SQL statements. They are listed here according to their functions.

Retrieving data using the SELECT statement

The SELECT statement tailors your query to gather data. You can use the SELECT statement to retrieve a specific row or retrieve data in a specific way.

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.

Related reference

"Creating a table using AS" on page 20
You can create a table from the result of a SELECT statement. To create this type of table, use the CREATE TABLE AS statement.

"Creating and using views" on page 31
A view can be used to access data in one or more tables or views. You create a view by using a SELECT statement.

Basic SELECT statement

The basic format and syntax of the SELECT statement consists of several required and optional clauses.

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). A SELECT statement can also be used by a cursor in a program. Finally, a SELECT statement can be prepared in a dynamic application.

Notes:

1. The SQL statements described in this section can be run on SQL tables and views, and database physical and logical files.

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

A SELECT statement can include the following:

1. The name of each column you want to include in the result.
2. The name of the table or view that contains the data.
3. A search condition to identify the rows that contain 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 the table EMPLOYEE.

The SELECT list can also contain expressions, including constants, special registers, and scalar fullselects. An AS clause can 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 follows.

Table 7. Results for query

LASTNAME	RAISE
NATZ	1421

Specifying a search condition using the WHERE clause

The WHERE clause specifies a search condition that identifies the row or rows that 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.

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, issue the following statement:

```
... 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'
```

The result of this statement is one row.

Table 8. Result table

DEPTNAME	MGRNO
INFORMATION CENTER	000030

If the search condition contains character, or UCS-2 or UTF-16 graphic column predicates, the sort sequence that is in effect when the query is run is applied to those predicates. If a sort sequence is not being used, character constants must be specified in uppercase or lowercase to match the column or expression they are being compared to.

Related concepts

"Sort sequences and normalization in SQL" on page 102
A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.

Related reference

"Defining complex search conditions" on page 52
In addition to the basic comparison predicates, such as = and >, a search condition can contain any of these predicates: BETWEEN, IN, EXISTS, IS NULL, and LIKE.

"Multiple search conditions within a WHERE clause" on page 54
You can qualify your request further by coding a search condition that includes several predicates.

Expressions in the WHERE clause:

An expression in a WHERE clause names or specifies something that you want to compare to something else.

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

A search condition can specify many 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.

To fully understand the WHERE clause, you need to know the order SQL evaluates search conditions and predicates, and compares the values of expressions. This topic is discussed in the DB2 for i5/OS SQL reference topic collection.

Related concepts

"Using subqueries" on page 94
You can use subqueries in a search condition as another way to select data. Subqueries can be used anywhere an expression can be used.

Related reference

"Defining complex search conditions" on page 52
In addition to the basic comparison predicates, such as = and >, a search condition can contain any of these predicates: BETWEEN, IN, EXISTS, IS NULL, and LIKE.

Expressions

Comparison operators:

SQL supports several comparison operators.

Comparison operator	Description
<> or ¬= or !=	Not equal to
=	Equal to
<	Less than
>	Greater than
<= or ¬> or !>	Less than or equal to (or not greater than)
> = or ¬< 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).

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 the department C01, you can say:

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

which is equivalent to:

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

GROUP BY clause

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.

Without a GROUP BY clause, the application of SQL aggregate 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.

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 can 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. Depending on the implementation selected by the database manager, the resulting groups might 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 or UTF-16 graphic columns, the sort sequence in effect when the query is run is applied to the grouping.

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

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

The result follows.

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 processes 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.

Related concepts

"Sort sequences and normalization in SQL" on page 102
A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.

Related reference

"ORDER BY clause" on page 45
The ORDER BY clause specifies the particular order in which you want selected rows returned. The order is sorted by ascending or descending collating sequence of a column's or an expression's value.

HAVING clause

The HAVING clause specifies a search condition for the groups selected by the 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 as you can specify in a WHERE clause. In addition, you can specify aggregate functions in a HAVING clause. For example, suppose that you want to retrieve the average salary of women in each department. To do this, use the AVG aggregate 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
```

The result follows.

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. These clauses make it a read-only cursor.

Predicates with arguments that are not aggregate functions can be coded in either WHERE or HAVING clauses. It is typically 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 or UTF-16 graphic columns, the sort sequence in effect when the query is run is applied to those predicates.

> **Related concepts**

> "Sort sequences and normalization in SQL" on page 102
> A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.

> **Related reference**

> "Using a cursor" on page 212
> When SQL runs a SELECT statement, the resulting rows comprise the result table. A cursor provides a way to access a result table.

ORDER BY clause

The ORDER BY clause specifies the particular order in which you want selected rows returned. The order is sorted by ascending or descending collating sequence of a column's or an expression's value.

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

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

The result follows.

LASTNAME	WORKDEPT
HAAS	A00
HEMMINGER	A00
KWAN	C01
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,FIRSTNME
       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 names of employees listed in alphabetic order, you can use this select-statement:

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

This select-statement can optionally be written as:

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

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
```

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 or UTF-16 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.

Related concepts

"Sort sequences and normalization in SQL" on page 102
A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.

Related reference

"GROUP BY clause" on page 42
The GROUP BY clause allows you to find the characteristics of groups of rows rather than individual rows.

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, will 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.

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 (SQLWARN3 will contain a "W"). When using the GET DIAGNOSTICS statement, the RETURNED_SQLSTATE item will have a value of '01503'.

Related reference

"Using a cursor" on page 212
When SQL runs a SELECT statement, the resulting rows comprise the result table. A cursor provides a way to access a result table.

Handling null values

A null value indicates the absence of a column value in a row. A null value is an unknown value; it is not the same as zero or all blanks.

Null values can be used as a condition in the WHERE and HAVING clauses. For example, a WHERE clause can specify a column that, for some rows, contains a null value. A basic 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 not less than, equal to, or greater than the value specified in the condition. The IS NULL predicate is used to check for null values. To select the values for all rows that contain a null value for the manager number, you can specify:

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

The result follows.

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 can change the WHERE clause like this:

```
WHERE MGRNO IS NOT NULL
```

Another predicate that is useful for comparing values that can contain the NULL value is the DISTINCT predicate. Comparing two columns using a normal equal comparison (COL1 = COL2) will be true if both columns contain an equal non-null value. If both columns are null, the result will be false because null is never equal to any other value, not even another null value. Using the DISTINCT predicate, null values are considered equal. So COL1 is NOT DISTINCT from COL2 will be true if both columns contain an equal non-null value and also when both columns are the null value.

For example, suppose that you want to select information from two tables that contain null values. The first table T1 has a column C1 with the following values.

C1
2
1
null

The second table T2 has a column C2 with the following values.

C2
2
null

Run the following SELECT statement:

```
SELECT *
    FROM T1, T2
    WHERE C1 IS DISTINCT FROM C2
```

The result follows.

C1	C2
1	2
1	-
2	-
-	2

For more information about the use of null values, see the DB2 for i5/OS SQL reference topic collection.

Special registers in SQL statements

You can specify certain special registers in SQL statements. A *special register*, for example, CURRENT DATE, contains information that can be referenced in SQL statements.

For locally run SQL statements, the special registers and their contents are shown in the following table.

Special registers	Contents
CURRENT CLIENT_ACCTNG	The accounting string for the client connection.
CURRENT CLIENT_APPLNAME	The application name for the client connection.
CURRENT CLIENT_PROGRAMID	The program ID for the client connection.
CURRENT CLIENT_USERID	The client user ID for the client connection.
CURRENT CLIENT_WRKSTNNAME	The workstation name for the client connection.

Special registers	Contents
CURRENT DATE CURRENT_DATE	The current date.
CURRENT DEBUG MODE	The debug mode to be used when creating or altering routines.
CURRENT DECFLOAT ROUNDING MODE	The rounding mode to be used when working with decimal floating point values.
CURRENT DEGREE	The number of tasks the database manager should run in parallel.
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.
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.

SESSION_USER USER	The runtime authorization identifier (user profile) of the job.
SYSTEM_USER	The authorization identifier (user profile) of the user connected to the database.

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 values for special registers are determined 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 the DB2 Multisystem topic collection.

Casting data types

Sometimes you need to cast or change the type of an expression to a different data type or to the same data type with a different length, precision, or scale.

For example, if you want to compare two columns of different types, such as a user-defined type based on a character and an integer, you can change the character to an integer or the integer to a character 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 specification to explicitly cast a data type to another data type. For example, if you have a column of dates (BIRTHDATE) defined as DATE and want to cast the column data type to CHARACTER with a fixed length of 10, enter the following:

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

You can also use the CAST specification to cast data types directly:

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

Related reference

Casting between data types

Date, time, and timestamp data types

Date, time, and timestamp are data types that are represented in an internal form not seen by an 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 function.
- A value returned by the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special register.
- A value of a character string in the ANSI/ISO standard date, time, or timestamp format, for example, **DATE** '1950-01-01'.
- 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.

Related reference

Data types

Specifying current date and time values:

You can specify a current date, time, or timestamp in an expression by using one of these special registers: CURRENT DATE, CURRENT TIME, and 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 that contains 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.

> **Related reference**
>
> Datetime arithmetic in SQL

Row change expressions

The ROW CHANGE TIMESTAMP and ROW CHANGE TOKEN expressions can be used to determine when a row was last changed.

To use a ROW CHANGE TIMESTAMP expression for a table, the table must be defined to have a row change timestamp column.

The following query can find all the orders that are at least four weeks old and can list when they were last modified:

```
SELECT ORDERNO, ROW CHANGE TIMESTAMP FOR ORDERS
    FROM ORDERS
    WHERE ORDER_DATE < CURRENT DATE - 4 WEEKS
```

The ROW CHANGE TOKEN expression can be used for both tables that have a row change timestamp and tables that do not. It represents a modification point for a row. If a table has a row change timestamp, it is derived from the timestamp. If a table does not have a row change timestamp, it is based on an internal modification time that is not row-based, so it is not as accurate as for a table that has a row change timestamp.

Handling 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 search condition of the select-statement. Some of the rows in the result table might be duplicate.

You can specify that you do not want any duplicates by using the DISTINCT keyword, followed by the list of expressions:

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

DISTINCT means that 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 might be returned. The sort sequence causes values that contain the same characters to be weighted the same. If 'MGR', 'Mgr', and 'mgr' are all in the same table, only one of these values is returned.

Related concepts

"Sort sequences and normalization in SQL" on page 102
A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.

Defining complex search conditions

In addition to the basic comparison predicates, such as = and >, a search condition can contain any of these predicates: BETWEEN, IN, EXISTS, IS NULL, and LIKE.

A search condition can include a scalar fullselect.

For character, or UCS-2 or UTF-16 graphic column predicates, the sort sequence is applied to the operands before evaluation of the predicates for BETWEEN, IN, EXISTS, and LIKE clauses.

You can also perform multiple search conditions.

- **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 can 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 can 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 can 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 can specify:

  `... WHERE EMPLOYEE.PHONE IS NULL`

- **LIKE** says you are interested in rows in which an expression 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 DB2 for i5/OS SQL reference topic collection 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 can 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 can 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 can specify:

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

The first and last percent characters in the LIKE string are interpreted as the normal LIKE percent characters. The combination '@%' is taken as the actual percent character.

Related concepts

"Using subqueries" on page 94
You can use subqueries in a search condition as another way to select data. Subqueries can be used anywhere an expression can be used.

"Sort sequences and normalization in SQL" on page 102
A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.

Related reference

"Specifying a search condition using the WHERE clause" on page 39
The WHERE clause specifies a search condition that identifies the row or rows that you want to retrieve, update, or delete.

"Expressions in the WHERE clause" on page 40
An expression in a WHERE clause names or specifies something that you want to compare to something else.

Predicates

Special considerations for LIKE:

Here are some considerations for using the LIKE predicate.

- 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 changes.
  - Obtain the same selection criteria and results as if a string constant were 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 do a search using the string pattern 'ABC%' in a varying length host variable, these are some of the values that can be returned:

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

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

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

Note: All returned values start with 'ABC' and end with at least 6 blanks. Blanks are used because the last 6 characters in the host variable are not assigned a specific value.

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

```
'ABCDEFGHIJ'   'ABCXXXXXX'  'ABCDE'       'ABCDD'
```

Multiple search conditions within a WHERE clause:

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, DISTINCT, IN, LIKE, EXISTS, IS NULL, and IS NOT NULL.

You can combine any two predicates with AND and OR. In addition, you can use the NOT keyword to specify that the search condition that you want 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, 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 can specify :

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

  Note: You can 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 the department E11 except those with a job code equal to analyst, you can 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 can 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

If you are combining multiple equal comparisons, you can write the predicate with the ANDs as shown in the following example:

```
...
  WHERE WORKDEPT = 'E11' AND EDLEVEL = 12 AND JOB = 'CLERK'
```

You can also compare two lists, for example:

```
...
  WHERE (WORKDEPT, EDLEVEL, JOB) = ('E11', 12, 'CLERK')
```

When two lists are used, the first item in the first list is compared to the first item in the second list, and so on through both lists. Thus, each list must contain the same number of entries. Using lists is identical to writing the query with AND. Lists can only be used with the equal and not equal comparison operators.

> **Related reference**
>
> "Specifying a search condition using the WHERE clause" on page 39
> The WHERE clause specifies a search condition that identifies the row or rows that you want to retrieve, update, or delete.

Using OLAP specifications

Online analytical processing (OLAP) specifications are used to return ranking numbers and row numbers for the result rows of a query. You can specify RANK, DENSE_RANK, and ROW_NUMBER.

Example: Ranking and row numbering

Suppose that you want a list of the top 10 salaries along with their ranking. The following query generates the ranking number for you:

```
SELECT EMPNO, SALARY,
       RANK() OVER(ORDER BY SALARY DESC),
       DENSE_RANK() OVER(ORDER BY SALARY DESC),
       ROW_NUMBER() OVER(ORDER BY SALARY DESC)
FROM EMPLOYEE
FETCH FIRST 10 ROWS ONLY
```

This query returns the following information.

Table 9. Results of the previous query

EMPNO	SALARY	RANK	DENSE_RANK	ROW_NUMBER
000010	52,750.00	1	1	1
000110	46,500.00	2	2	2
200010	46,500.00	2	2	3
000020	41,250.00	4	3	4
000050	40,175.00	5	4	5
000030	38,250.00	6	5	6
000070	36,170.00	7	6	7
000060	32,250.00	8	7	8
000220	29,840.00	9	8	9
200220	29,840.00	9	8	10

In this example, the SALARY descending order with the top 10 returned. The RANK column shows the relative ranking of each salary. Notice that there are two rows with the same salary at position 2. Each of those rows is assigned the same rank value. The following row is assigned the value of 4. RANK returns

a value for a row that is one more than the total number of rows that precede that row. There are gaps in the numbering sequence whenever there are duplicates.

In contrast, the DENSE_RANK column shows a value of 3 for the row directly after the duplicate rows. DENSE_RANK returns a value for a row that is one more than the number of distinct row values that precede it. There will never be gaps in the numbering sequence.

ROW_NUMBER returns a unique number for each row. For rows that contain duplicate values according to the specified ordering, the assignment of a row number is arbitrary; the row numbers could be assigned in a different order for the duplicate rows when the query is run another time.

Example: Ranking groups

Suppose that you want to find out which department has the highest average salary. The following query groups the data by department, determines the average salary for each department, and ranks the resulting averages.

```
SELECT WORKDEPT, INT(AVG(SALARY)) AS AVERAGE,
        RANK() OVER(ORDER BY AVG(SALARY) DESC) AS AVG_SALARY
    FROM EMPLOYEE
    GROUP BY  WORKDEPT
```

This query returns the following information.

Table 10. Results of previous query

WORKDEPT	AVERAGE	AVG_SALARY
B01	41,250	1
A00	40,850	2
E01	40,175	3
C01	29,722	4
D21	25,668	5
D11	25,147	6
E21	24,086	7
E11	21,020	8

Example: Ranking within a department

Suppose that you want a list of employees along with how their bonus ranks within their department. Using the PARTITION BY clause, you can specify groups that are to be numbered separately.

```
SELECT LASTNAME, WORKDEPT, BONUS,
        DENSE_RANK() OVER(PARTITION BY WORKDEPT ORDER BY BONUS DESC)
          AS BONUS_RANK_IN_DEPT
    FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%'
```

This query returns the following information.

Table 11. Results of the previous query

LASTNAME	WORKDEPT	BONUS	BONUS_RANK_in_DEPT
GEYER	E01	800.00	1
HENDERSON	E11	600.00	1
SCHNEIDER	E11	500.00	2
SCHWARTZ	E11	500.00	2

Table 11. Results of the previous query (continued)

LASTNAME	WORKDEPT	BONUS	BONUS_RANK_in_DEPT
SMITH	E11	400.00	3
PARKER	E11	300.00	4
SETRIGHT	E11	300.00	4
SPRINGER	E11	300.00	4
SPENSER	E21	500.00	1
LEE	E21	500.00	1
GOUNOT	E21	500.00	1
WONG	E21	500.00	1
ALONZO	E21	500.00	1
MENTA	E21	400.00	2

Example: Ranking and ordering by table expression results

Suppose that you want to find the top five employees whose salaries are the highest along with their department names. The department name is in the *department* table, so a join operation is needed. Because ordering is already being done in the nested table expression, that ordering can also be used for determining the ROW_NUMBER value. The ORDER BY ORDER OF *table* clause is used to do this.

```
SELECT ROW_NUMBER() OVER(ORDER BY ORDER OF EMP),
           EMPNO, SALARY, DEPTNO, DEPTNAME
  FROM (SELECT EMPNO, WORKDEPT, SALARY
           FROM EMPLOYEE
           ORDER BY SALARY DESC
           FETCH FIRST 5 ROWS ONLY) EMP,
        DEPARTMENT
  WHERE DEPTNO = WORKDEPT
```

This query returns the following information.

Table 12. Results of the previous query

ROW_NUMBER	EMPNO	SALARY	DEPTNO	DEPTNAME
1	000010	52,750.00	A00	SPIFFY COMPUTER SERVICE DIV.
2	000110	46,500.00	A00	SPIFFY COMPUTER SERVICE DIV.
3	200010	46,500.00	A00	SPIFFY COMPUTER SERVICE DIV.
4	000020	41,250.00	B01	PLANNING
5	000050	40,175.00	E01	SUPPORT SERVICES

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 for i5/OS: inner join, left outer join, right outer join, left exception join, right exception join, and cross join.

Usage notes on join operations

When you join two or more tables, consider the following items:

- 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. However, the USING clause can be used in a join to allow you to identify columns that exist in both tables without specifying table names.
- 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, or UCS-2 or UTF-16 graphic columns being joined.

Inner join:

An inner join 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 do not appear in the result table.

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 the 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:

To use the WHERE clause to perform the same join as you perform using the INNER JOIN syntax, enter 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.

Joining data with the USING clause:

You can use the USING clause for a shorthand way of defining join conditions. The USING clause is equivalent to a join condition where each column from the left table is compared to a column with the same name in the right table.

For example, look at the USING clause in this statement:

```
SELECT EMPNO, ACSTDATE
      FROM CORPDATA.PROJACT INNER JOIN CORPDATA.EMPPROJACT
      USING (PROJNO, ACTNO)
      WHERE ACSDATE > '1982-12-31';
```

The syntax in this statement is valid and equivalent to the join condition in the following statement:

```
SELECT EMPNO, ACSTDATE
      FROM CORPDATA.PROJACT INNER JOIN CORPDATA.EMPPROJACT
          ON CORPDATA.PROJACT.PROJNO = CORPDATA.EMPPROJACT.PROJNO AND
             CORPDATA.PROJACT.ACTNO = CORPDATA.EMPPROJACT.ACTNO
      WHERE ACSTDATE > '1982-12-31';
```

Left outer join:

A left outer join returns all the rows that an inner join returns plus one row for each of the other rows in the first table that do 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	-

EMPNO	LASTNAME	PROJNO
000190	WALKER	-
000250	SMITH	AD3112
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-
200170	YAMAMOTO	-
200280	SCHWARTZ	-
200310	SPRINGER	-
200330	WONG	-

Note: 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 returns all the rows that an inner join returns plus one row for each of the other rows in the second table that do 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 can 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 following output.

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

EMPNO	LASTNAME	PROJNO
200310	SPRINGER	-
200330	WONG	-

An exception join can also be written as a subquery using the NOT EXISTS predicate. The previous query can 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, also known as a Cartesian Product join, returns 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 that the following tables exist.

Table 13. Table A

ACOL1	ACOL2
A1	AA1
A2	AA2
A3	AA3

Table 14. 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

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

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 nonmatching rows from both tables.

Suppose that you want to find all employees and all of their projects. You want to see those employees that are not currently in charge of a project as well as any projects that do not have an employee assigned. The following query returns a list of all employees whose names are greater than 'S', along with their assigned project numbers:

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

Because there are no projects without an assigned employee, the query returns the same rows as a left outer join. Here are the results.

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	-

Multiple join types in one statement:

Sometimes you need to join more than two tables to produce the result that you want.

If you want to return all the employees, their department names, and the projects they are responsible for, if any, you need to join the EMPLOYEE table, the DEPARTMENT table, and the PROJECT table 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'
```

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	-

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 follows.

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

MGRNO	DEPTNO	MAXSAL
000090	E11	29750.00
000100	E21	26150.00

Common table expressions can be specified before 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 follows.

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

Suppose that 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 recursive queries

Some applications work with data that is recursive in nature. To query this type of data, you can use a recursive common table expression or a recursive view.

For example, a Bill of Materials (BOM) application works with the expansion of parts and its component subparts. For example, a chair might be made of a seat unit and a leg assembly. The seat unit might consist of a seat and two arms. Each of these parts can be further broken down into its subparts until there is a list of all the parts needed to build a chair.

In the following trip planner examples, airline flights and train connections are used to find transportation paths between cities. The following table definitions and data are used in the examples.

```
CREATE TABLE FLIGHTS (DEPARTURE CHAR(20),
                      ARRIVAL CHAR(20),
                      CARRIER CHAR(15),
                      FLIGHT_NUMBER CHAR(5),
                      PRICE INT)


INSERT INTO FLIGHTS VALUES('New York', 'Paris', 'Atlantic', '234', 400)
INSERT INTO FLIGHTS VALUES('Chicago', 'Miami', 'NA Air', '2334', 300)
INSERT INTO FLIGHTS VALUES('New York', 'London', 'Atlantic', '5473', 350)
INSERT INTO FLIGHTS VALUES('London', 'Athens'  , 'Mediterranean', '247', 340)
INSERT INTO FLIGHTS VALUES('Athens', 'Nicosia' , 'Mediterranean', '2356', 280)
INSERT INTO FLIGHTS VALUES('Paris', 'Madrid' , 'Euro Air',  '3256', 380)
INSERT INTO FLIGHTS VALUES('Paris', 'Cairo' , 'Euro Air', '63', 480)
INSERT INTO FLIGHTS VALUES('Chicago', 'Frankfurt', 'Atlantic', '37', 480)
INSERT INTO FLIGHTS VALUES('Frankfurt', 'Moscow', 'Asia Air', '2337', 580)
INSERT INTO FLIGHTS VALUES('Frankfurt', 'Beijing', 'Asia Air',  '77', 480)
INSERT INTO FLIGHTS VALUES('Moscow', 'Tokyo', 'Asia Air', '437', 680)
INSERT INTO FLIGHTS VALUES('Frankfurt', 'Vienna', 'Euro Air', '59', 200)
INSERT INTO FLIGHTS VALUES('Paris', 'Rome', 'Euro Air', '534', 340)
INSERT INTO FLIGHTS VALUES('Miami', 'Lima', 'SA Air', '5234', 530)
INSERT INTO FLIGHTS VALUES('New York', 'Los Angeles', 'NA Air', '84', 330)
INSERT INTO FLIGHTS VALUES('Los Angeles', 'Tokyo', 'Pacific Air', '824', 530)
INSERT INTO FLIGHTS VALUES('Tokyo', 'Hong Kong', 'Asia Air', '94', 330)
INSERT INTO FLIGHTS VALUES('Washington', 'Toronto', 'NA Air', '104', 250)

CREATE TABLE TRAINS(DEPARTURE CHAR(20),
                    ARRIVAL CHAR(20),
                    RAILLINE CHAR(15),
                    TRAIN CHAR(5),
                    PRICE INT)

INSERT INTO TRAINS VALUES('Chicago', 'Washington', 'UsTrack', '323', 90)
INSERT INTO TRAINS  VALUES('Madrid', 'Barcelona', 'EuroTrack', '5234', 60)
INSERT INTO TRAINS  VALUES('Washington' , 'Boston' , 'UsTrack', '232', 50)
```

Now that the tables are set up, the data can be queried to find information about the airline network. Suppose you want to find out what cities you can fly to if you start in Chicago, and how many separate flights it will take to get there. The following query shows you that information.

```
WITH destinations (origin, departure, arrival, flight_count) AS
    (SELECT a.departure, a.departure, a.arrival, 1
            FROM flights a
```

```
            WHERE a.departure = 'Chicago'
      UNION ALL
      SELECT r.origin, b.departure, b.arrival, r.flight_count + 1
            FROM destinations r, flights b
            WHERE r.arrival = b.departure)
SELECT origin, departure, arrival, flight_count
    FROM destinations
```

This query returns the following information.

Table 15. Results of the previous query

ORIGIN	DEPARTURE	ARRIVAL	FLIGHT_COUNT
Chicago	Chicago	Miami	1
Chicago	Chicago	Frankfurt	1
Chicago	Miami	Lima	2
Chicago	Frankfurt	Moscow	2
Chicago	Frankfurt	Beijing	2
Chicago	Frankfurt	Vienna	2
Chicago	Moscow	Tokyo	3
Chicago	Tokyo	Hong Kong	4

This recursive query is written in two parts. The first part of the common table expression is called the *intialization fullselect*. It selects the first rows for the result set of the common table expression. In this example, it selects the two rows in the *flights* table that get you directly to another location from Chicago. It also initializes the number of flight legs to one for each row it selects.

The second part of the recursive query joins the rows from the current result set of the common table expression with other rows from the original table. It is called the *iterative fullselect*. This is where the recursion is introduced. Notice that the rows that have already been selected for the result set are referenced by using the name of the common table expression as the table name and the common table expression result column names as the column names.

In this recursive part of the query, any rows from the original table that you can get to from each of the previously selected arrival cities are selected. A previously selected row's arrival city becomes the new departure city. Each row from this recursive select increments the flight count to the destination by one more flight. As these new rows are added to the common table expression result set, they are also fed into the iterative fullselect to generate more result set rows. In the data for the final result, you can see that the total number of flights is actually the total number of recursive joins (plus 1) it took to get to that arrival city.

A recursive view looks very similar to a recursive common table expression. You can write the previous recursive common table expression as a recursive view like this:

```
CREATE VIEW destinations (origin, departure, arrival, flight_count) AS
    SELECT departure, departure, arrival, 1
          FROM flights
          WHERE departure = 'Chicago'
    UNION ALL
    SELECT r.origin, b.departure, b.arrival, r.flight_count + 1
          FROM destinations r, flights b
          WHERE r.arrival = b.departure)
```

The interactive fullselect part of this view definition refers to the view itself. Selection from this view returns the same rows as you get from the previous recursive common table expression.

Example: Two starting cities

Now, to make the query a bit more complicated, suppose you are willing to fly from either Chicago or New York, and you want to know where you could go and how much it would cost.

```
WITH destinations (departure, arrival, connections, cost) AS
    (SELECT a.departure, a.arrival, 0, price
         FROM flights a
         WHERE a.departure = 'Chicago' OR
             a.departure = 'New York'
     UNION ALL
     SELECT r.departure, b.arrival, r.connections + 1,
             r.cost + b.price
         FROM destinations r, flights b
         WHERE r.arrival = b.departure)
SELECT departure, arrival, connections, cost
     FROM destinations
```

This query returns the following information.

Table 16. Results of the previous query

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Frankfurt	0	480
New York	Paris	0	400
New York	London	0	350
New York	Los Angeles	0	330
Chicago	Lima	1	830
Chicago	Moscow	1	1,060
Chicago	Beijing	1	960
Chicago	Vienna	1	680
New York	Madrid	1	780
New York	Cairo	1	880
New York	Rome	1	740
New York	Athens	1	690
New York	Tokyo	1	860
Chicago	Tokyo	2	1,740
New York	Nicosia	2	970
New York	Hong Kong	2	1,190
Chicago	Hong Kong	3	2,070

For each returned row, the results show the starting departure city and the final destination city. It counts the number of connections needed rather than the total number of flight and adds up the total cost for all the flights.

Example: Two tables used for recursion

Now, suppose you start in Chicago but add in transportation by railway in addition to the airline flights, and you want to know which cities you can go to.

The following query returns that information:

```
WITH destinations (departure, arrival, connections, flights, trains, cost) AS
   (SELECT f.departure, f.arrival, 0, 1, 0, price
          FROM  flights f
          WHERE f.departure = 'Chicago'
   UNION ALL
   SELECT t.departure, t.arrival, 0, 0, 1, price
          FROM trains t
          WHERE t.departure = 'Chicago'
   UNION ALL
   SELECT r.departure, b.arrival, r.connections + 1 , r.flights + 1, r.trains,
             r.cost + b.price
          FROM  destinations r, flights b
          WHERE r.arrival = b.departure
   UNION ALL
   SELECT r.departure, c.arrival, r.connections + 1 ,
             r.flights, r.trains + 1, r.cost + c.price
          FROM destinations r, trains c
          WHERE r.arrival = c.departure)
SELECT departure, arrival, connections, flights, trains, cost
       FROM destinations
```

This query returns the following information.

Table 17. Results of the previous query

DEPARTURE	ARRIVAL	CONNECTIONS	FLIGHTS	TRAINS	COST
Chicago	Miami	0	1	0	300
Chicago	Frankfurt	0	1	0	480
Chicago	Washington	0	0	1	90
Chicago	Lima	1	2	0	830
Chicago	Moscow	1	2	0	1,060
Chicago	Beijing	1	2	0	960
Chicago	Vienna	1	2	0	680
Chicago	Toronto	1	1	1	340
Chicago	Boston	1	0	2	140
Chicago	Tokyo	2	3	0	1,740
Chicago	Hong Kong	3	4	0	2,070

In this example, there are two parts of the common table expression that provide initialization values to
the query: one for flights and one for trains. For each of the result rows, there are two recursive
references to get from the previous arrival location to the next possible destination: one for continuing by
air, the other for continuing by train. In the final results, you would see how many connections are
needed and how many airline or train trips can be taken.

Example: DEPTH FIRST and BREADTH FIRST options

The two examples here show the difference in the result set row order based on whether the recursion is
processed depth first or breadth first.

Note: The search clause is not supported for recursive views. You can define a view that contains a
recursive common table expression to get this function.

The option to determine the result using breadth first or depth first is a recursive relationship sort based
on the recursive join column specified for the SEARCH BY clause. When the recursion is handled breadth

first, all children are processed first, then all grandchildren, then all great grandchildren. When the recursion is handled depth first, the full recursive ancestry chain of one child is processed before going to the next child.

In both of these cases, you specify an extra column name that is used by the recursive process to keep track of the depth first or breadth first ordering. This column must be used in the ORDER BY clause of the outer query to get the rows back in the specified order. If this column is not used in the ORDER BY, the DEPTH FIRST or BREADTH FIRST processing option is ignored.

The selection of which column to use for the SEARCH BY column is important. To have any meaning in the result, it must be the column that is used in the iterative fullselect to join from the initialization fullselect. In this example, ARRIVAL is the column to use.

The following query returns that information:

```
WITH destinations (departure, arrival, connections, cost) AS
    (SELECT f.departure, f.arrival, 0, price
            FROM flights f
            WHERE f.departure = 'Chicago'
     UNION ALL
     SELECT r.departure, b.arrival, r.connections + 1,
                r.cost + b.price
            FROM destinations r, flights b
            WHERE r.arrival = b.departure)
    SEARCH DEPTH FIRST BY arrival SET ordcol
SELECT *
    FROM destinations
    ORDER BY ordcol
```

This query returns the following information.

Table 18. Results of the previous query

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Lima	1	830
Chicago	Frankfurt	0	480
Chicago	Moscow	1	1,060
Chicago	Tokyo	2	1,740
Chicago	Hong Kong	3	2,070
Chicago	Beijing	1	960
Chicago	Vienna	1	680

In this result data, you can see that all destinations that are generated from the Chicago-to-Miami row are listed before the destinations from the Chicago-to-Frankfort row.

Next, you can run the same query but request the result to be ordered breadth first.

```
WITH destinations (departure, arrival, connections, cost) AS
    (SELECT f.departure, f.arrival, 0, price
            FROM flights f
            WHERE f.departure='Chicago'
     UNION ALL
     SELECT r.departure, b.arrival, r.connections + 1,
                r.cost + b.price
            FROM destinations r, flights b
            WHERE r.arrival = b.departure)
```

```
      SEARCH BREADTH FIRST BY arrival SET ordcol
SELECT *
  FROM destinations
  ORDER BY ordcol
```

This query returns the following information.

Table 19. Results of the previous query

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Frankfurt	0	480
Chicago	Lima	1	830
Chicago	Moscow	1	1,060
Chicago	Beijing	1	960
Chicago	Vienna	1	680
Chicago	Tokyo	2	1,740
Chicago	Hong Kong	3	2,070

In this result data, you can see that all the direct connections from Chicago are listed before the connecting flights. The data is identical to the results from the previous query, but in a breadth first order.

Example: Cyclic

The key to any recursive process, whether it is a recursive programming algorithm or querying recursive data, is that the recursion must be finite. If not, you will get into a never ending loop. The CYCLE option allows you to safeguard against cyclic data. Not only will it terminate repeating cycles but it also allows you to optionally output a cycle mark indicator that may lead you to find cyclic data.

Note: The cycle clause is not supported for recursive views. You can define a view that contains a recursive common table expression to get this function.

For a final example, suppose we have a cycle in the data. By adding one more row to the table, there is now a flight from Cairo to Paris and one from Paris to Cairo. Without accounting for possible cyclic data like this, it is quite easy to generate a query that will go into an infinite loop processing the data.

The following query returns that information:
```
INSERT INTO FLIGHTS VALUES('Cairo', 'Paris', 'Euro Air', '1134', 440)


WITH destinations (departure, arrival, connections, cost, itinerary) AS
    (SELECT f.departure, f.arrival, 1, price,
             CAST(f.departure CONCAT f.arrival AS VARCHAR(2000))
          FROM flights f
          WHERE f.departure = 'New York'
     UNION ALL
     SELECT r.departure, b.arrival, r.connections + 1 ,
             r.cost + b.price, CAST(r.itinerary CONCAT b.arrival AS VARCHAR(2000))
          FROM destinations r, flights b
          WHERE r.arrival = b.departure)
    CYCLE arrival SET cyclic_data TO '1' DEFAULT '0'
SELECT departure, arrival, itinerary, cyclic_data
      FROM destinations
      ORDER BY cyclic_data
```

This query returns the following information.

Table 20. Results of the previous query

DEPARTURE	ARRIVAL	ITINERARY				CYCLIC_DATA
New York	Paris	New York	Paris			0
New York	London	New York	London			0
New York	Los Angeles	New York	Los Angeles			0
New York	Madrid	New York	Paris	Madrid		0
New York	Cairo	New York	Paris	Cairo		0
New York	Rome	New York	Paris	Rome		0
New York	Athens	New York	London	Athens		0
New York	Tokyo	New York	Los Angeles	Tokyo		0
New York	Paris	New York	Paris	Cairo	Paris	1
New York	Nicosia	New York	London	Athens	Nicosia	0
New York	Hong Kong	New York	Los Angeles	Tokyo	Hong Kong	0

In this example, the ARRIVAL column is defined in the CYCLE clause as the column to use for detecting a cycle in the data. When a cycle is found, a special column, CYCLIC_DATA in this case, is set to the character value of '1' for the cycling row in the result set. All other rows will contain the default value of '0'. When a cycle on the ARRIVAL column is found, processing will not proceed any further in the data so the infinite loop will not happen. To see if your data actually has a cyclic reference, the CYCLIC_DATA column can be referenced in the outer query.

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 can use different clauses and techniques 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.EMPPROJACT
   WHERE PROJNO = 'MA2112' OR
      PROJNO = 'MA2113' OR
      PROJNO = 'AD3111'
 ORDER BY EMPNO
```

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'
```

The query returns the following 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'
```

The query returns another interim result table.

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

The query returns 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.

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.

Related concepts

"Sort sequences and normalization in SQL" on page 102
A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.

Related reference

"Creating and using views" on page 31
A view can be used to access data in one or more tables or views. You create a view by using a SELECT statement.

Specifying the UNION ALL keyword:

If you want to keep duplicates in the result of a UNION operation, specify the UNION ALL keyword instead of just UNION.

This topic uses the same steps and example as "Using the UNION keyword to combine subselects" on page 71.

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
```

The query returns an ordered result table that includes duplicates.

EMPNO
000060
000150
000150
000150
000160
000160
000170
000170
000170
000170
000180
000180
000190

EMPNO
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.EMPPROJACT
```

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.EMPPROJACT)
```

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.

Using the EXCEPT keyword

The EXCEPT keyword returns the result set of the first subselect minus any matching rows from the second subselect.

Suppose that you want to find a list of employee numbers that includes people in department D11 minus those people whose assignments include projects MA2112, MA2113, and AD3111.

This query returns all of the people in department D11 who are *not* working on projects MA2112, MA2113, and AD3111:

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

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

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'
```

This query returns 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'
```

This query returns another interim result table.

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

EMPNO from CORPDATA.EMPPROJACT
000170
000190
000150
000160
000180
000170
000210
000210

Step 3. SQL takes the first interim result table, removes all of the rows that also appear in the second interim result table, removes duplicate rows, and orders the result:

```
SELECT EMPNO
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11'
EXCEPT
SELECT EMPNO
     FROM CORPDATA.EMPPROJACT
     WHERE PROJNO='MA2112' OR
               PROJNO= 'MA2113' OR
               PROJNO= 'AD3111'
ORDER BY EMPNO
```

This query returns a combined result table with values in ascending sequence.

EMPNO
000060
000200
000220
200170
200220

Using the INTERSECT keyword

The INTERSECT keyword returns a combined result set that consists of all of the rows existing in both result sets.

Suppose that you want to find a list of employee numbers that includes people in department D11 and people whose assignments include projects MA2112, MA2113, and AD3111.

The INTERSECT operation returns all of the employee numbers that exist in both result sets. In other words, this query returns all of the people in department D11 who are also working on projects MA2112, MA2113, and AD3111:

```
SELECT EMPNO
   FROM CORPDATA.EMPLOYEE
   WHERE WORKDEPT = 'D11'
 INTERSECT
 SELECT EMPNO
   FROM CORPDATA.EMPPROJACT
   WHERE PROJNO = 'MA2112' OR
       PROJNO = 'MA2113' OR
       PROJNO = 'AD3111'
 ORDER BY EMPNO
```

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'
```

This query returns 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'
```

This query returns another interim result table.

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

EMPNO from CORPDATA.EMPPROJACT
000180
000170
000210
000210

Step 3. SQL takes the first interim result table, compares it to the second interim result table, and returns the rows that exist in both tables minus any duplicate rows, and orders the results.

```
SELECT EMPNO
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11'
INTERSECT
SELECT EMPNO
     FROM CORPDATA.EMPPROJACT
     WHERE PROJNO='MA2112' OR
               PROJNO= 'MA2113' OR
               PROJNO= 'AD3111'
ORDER BY EMPNO
```

This query returns a combined result table with values in ascending sequence.

EMPNO
000150
000160
000170
000180
000190
000210

Data retrieval errors

Use this information to understand how SQL handles errors that occur when retrieving data.

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' or sets RETURNED_SQLSTATE to '01004' in the SQL diagnostics area.
- 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.

Data mapping errors include:

- +138 - Argument of the substringing 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.
- +364 - DECFLOAT arithmetic error.
- +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.

For data mapping errors on a multi-row FETCH, each mapping error reported as a warning SQLSTATE will have a separate condition area in the SQL diagnostics area. Note that SQL stops on the first error, so only one mapping error that is reported as an error SQLSTATE will be returned in the SQL diagnostics area.

For all other SQL statements, only the last warning SQLSTATE will be reported in the SQL diagnostics area.

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 cannot be converted.
  - The expression is a decimal column and a numeric value that is not valid is detected.

Inserting rows using the INSERT statement

To add a single row or multiple rows to a table or view, use a form of 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 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.
- Specifying the blocked form of the INSERT statement to add multiple rows.

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 **scalar fullselect** 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 fixed length binary columns, the default is hexadecimal zeros.
- For varying length character, graphic, or binary columns and for 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.

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) in the SQLCA. It is also available from the ROW_COUNT diagnostics item in the GET DIAGNOSTICS statement.

Related reference

INSERT

Inserting rows using the VALUES clause

You use the VALUES clause in the INSERT statement to insert a single row or multiple rows into a table.

An example of this is to insert a new row into the DEPARTMENT table. The columns for the new row are as follows:

- Department number (DEPTNO) is 'E31'
- Department name (DEPTNAME) is 'ARCHITECTURE'
- Manager number (MGRNO) is '00390'
- Reports to (ADMRDEPT) department 'E01'

The INSERT statement for this new row is as follows:
```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT)
   VALUES('E31', 'ARCHITECTURE', '00390', 'E01')
```

You can also insert multiple rows into a table using the VALUES clause. The following example inserts two rows into the PROJECT table. Values for the Project number (PROJNO) , Project name (PROJNAME), Department number (DEPTNO), and Responsible employee (RESPEMP) are given in the values list. The value for the Project start date (PRSTDATE) uses the current date. The rest of the columns in the table that are not listed in the column list are assigned their default value.
```
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
   VALUES('HG0023', 'NEW NETWORK', 'E11', '200280', CURRENT DATE),
         ('HG0024', 'NETWORK PGM', ''E11", '200310', CURRENT DATE)
```

Inserting rows using a select-statement

You can use a select-statement within an INSERT statement to insert zero, one, or more rows into a table from the result table of the select-statement.

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. 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.EMPPROJACT
```

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, 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.

You should consider the following when inserting multiple rows into a table:

Notes:

1. The number of columns implicitly or explicitly listed in the INSERT statement must equal the number of columns listed in the select-statement.
2. 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.
3. 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. This value is also available from the ROW_COUNT diagnostics item in the GET DIAGNOSTICS statement.
4. 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 before the error remain in the table.

Inserting multiple rows using the blocked INSERT statement

Using a blocked INSERT statement, you can insert multiple rows into a table with a single INSERT 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 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 System i products.

> **Related concepts**
>
> Embedded SQL programming

Inserting data into tables with referential constraints

When you insert data into tables with referential constraints, you need to consider these rules.

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).

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 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 is unsuccessful because 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')
```

Inserting values 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, a table 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 can 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.

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.

> **Related reference**
>
> "Creating and altering an identity column" on page 23
> Every time a row is added to a table with an identity column, the identity column value for the new row is generated by the system.
>
> Scalar functions

Selecting inserted values

You can retrieve the values for rows that are being inserted by specifying the INSERT statement in the FROM clause of a SELECT statement.

When you insert one or more rows into a table, you can select the result rows of the insert operation. These rows include any of the following values:

- The value of any generated column, such as identity, ROWID, or row change timestamp columns
- Any default values used for columns
- All values for all rows inserted by a multiple-row insert operation
- Values that were changed by a before insert trigger

The following example uses a table defined as follows:

```
CREATE TABLE EMPSAMP
   (EMPNO     INTEGER GENERATED ALWAYS AS IDENTITY,
    NAME      CHAR(30),
    SALARY    DECIMAL(10,2),
    DEPTNO    SMALLINT,
    LEVEL     CHAR(30),
    HIRETYPE  VARCHAR(30) NOT NULL DEFAULT 'New Employee',
    HIREDATE  DATE NOT NULL WITH DEFAULT)
```

To insert a row for a new employee and see the values that were used for EMPNO, HIRETYPE, and HIREDATE, use the following statement:

```
SELECT EMPNO, HIRETYPE, HIREDATE
  FROM FINAL TABLE ( INSERT INTO EMPSAMP (NAME, SALARY, DEPTNO, LEVEL)
                                 VALUES('Mary Smith', 35000.00, 11, 'Associate'))
```

The returned values are the generated value for EMPNO, 'New Employee' for HIRETYPE, and the current date for HIREDATE.

Changing data in a table using the UPDATE statement

To update data in a table or view, use the UPDATE statement.

With the UPDATE statement, you can change the value of one or more columns in each row that meets 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 meet 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 ...
```

Suppose that an employee is relocated. To update the CORPDATA.EMPLOYEE table to reflect the move, run the following statement:

```
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 that you want to update. The SET clause names the columns that you want updated and provides the values that you want them changed to. You can specify the following types of values:

- 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 fullselect.** 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 UPDATE statement 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 meet 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. This value is also available from the ROW_COUNT diagnostics item in the GET DIAGNOSTICS statement.

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'
```

You can also write this UPDATE statement by specifying all of the columns and then all of the values:

```
UPDATE EMPLOYEE
  SET (WORKDEPT, PHONENO, JOB)
      = ('D11', '7213', 'DESIGNER')
  WHERE EMPNO = '000270'
```

   Related reference

   UPDATE

Updating a table using a scalar-subselect

Using a scalar-subselect, you can update one or more columns in a table with 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 can be used to update a list of columns with multiple values returned from a single select.

Updating a table with rows from another table

You can update an entire row in one table with values from a row in another table.

Suppose that a master class schedule table 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 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 occur in the following order:

1. A *before trigger* is fired before the UPDATE and before a constraint rule of RESTRICT or NO ACTION.

2. An *after trigger* is 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.

> **Related reference**
>
> "Journaling" on page 111
> The DB2 for i5/OS journal support provides an audit trail and forward and backward recovery.
>
> "Commitment control" on page 112
> The DB2 for i5/OS commitment control support provides a means for processing a group of database changes, such as update, insert, or delete operations or data definition language (DDL) operations, as a single unit of work (also referred to as a *transaction*).

Examples: UPDATE rules:

These examples illustrate the UPDATE rules for tables with referential constraints.

For example, you cannot update a department number from the DEPARTMENT table if the department is still responsible for a project that is described by a dependent row in the PROJECT table.

The following UPDATE statement fails because the PROJECT table has rows that are dependent on DEPARTMENT.DEPTNO that has a value of 'D01' (the row targeted by the WHERE statement). If this UPDATE statement were to be 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. Because D00 is not a value of the primary key DEPTNO in DEPARTMENT, the statement fails.

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 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'
```

Related reference

"Creating and altering an identity column" on page 23
Every time a row is added to a table with an identity column, the identity column value for the new row is generated by the system.

Updating data as it is retrieved from a table

You can update rows of data as you retrieve them by using a cursor.

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 21. Updating a table

Scrollable Cursor SQL Statement	Comments

EXEC SQL
  DECLARE THISEMP **DYNAMIC SCROLL CURSOR FOR**
  SELECT EMPNO, WORKDEPT, BONUS
    FROM CORPDATA.EMPLOYEE
    WHERE WORKDEPT = 'D11'
    FOR UPDATE OF BONUS
 END-EXEC.	

EXEC SQL **OPEN** THISEMP END-EXEC.	

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.	

Table 21. Updating a table (continued)

Scrollable Cursor SQL Statement	Comments

```
CLOSE-THISEMP.
EXEC SQL
  CLOSE THISEMP
END-EXEC.
```

Related reference

"Using a cursor" on page 212
When SQL runs a SELECT statement, the resulting rows comprise the result table. A cursor provides a way to access a result 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. The DELETE statement 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 from the table. The DELETE statement looks like this:

```
DELETE FROM table-name
  WHERE search-condition ...
```

For example, suppose that department D11 is moved to another site. You 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.

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) of the SQLCA. This value is also available from the ROW_COUNT diagnostics item in the GET DIAGNOSTICS statement.

Related reference

DROP

DELETE

Removing rows from tables with referential constraints

If a table has a primary key but no dependents, or if a table has only foreign keys but no primary key, the DELETE statement operates the same way as it does for tables without referential constraints. If a table has a primary key and dependent tables, the DELETE statement operates according to the delete rules specified for the table.

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 operation 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 descendents 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 descendents 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 SQLERRD(3) value is also available from the ROW_COUNT item in the GET DIAGNOSTICS statement. The SQLERRD(5) value is available from the DB2_ROW_COUNT_SECONDARY item.

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 occur in the following order:

1. A *before trigger* is fired before the DELETE and before a constraint rule of RESTRICT or NO ACTION.
2. An *after trigger* is fired after a constraint rule of RESTRICT, but before a NO ACTION rule.

Example: DELETE rules:

Suppose that 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 in the "DB2 for i5/OS sample tables" on page 277, 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 following sample data reflects the null value. The results 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 delete operations in the DEPARTMENT table because no department reported to department 'E11'.

Below are the 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
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

Using subqueries

You can use subqueries in a search condition as another way to select data. Subqueries can be used anywhere an expression can be used.

Conceptually, a subquery is evaluated whenever a new row or a 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**.

   Related reference
   "Expressions in the WHERE clause" on page 40
   An expression in a WHERE clause names or specifies something that you want to compare to
   something else.

"Defining complex search conditions" on page 52
In addition to the basic comparison predicates, such as = and >, a search condition can contain any of these predicates: BETWEEN, IN, EXISTS, IS NULL, and LIKE.

Subqueries in SELECT statements

Subqueries further refine your search conditions 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 can issue just one SQL statement to retrieve the employee numbers, names, and job codes for employees who work on the 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')
```

The result in an interim result table follows.

EMPNO from CORPDATA.EMPPROJACT
000010
000110

Step 2: The interim result table then serves as a list in the search condition of the outer-level SELECT statement. 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	LUCCHESSI	SALESREP

Subqueries and search conditions:

A subquery can be part of a search condition. The search condition is in the form of *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 that contains 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 can 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.

Usage notes on subqueries:

Here are some considerations for using subqueries to refine your search conditions.

1. When nesting SELECT statements, you can use as many as you need to satisfy your requirements (1 to 255 subqueries), although performance is slower for each additional subquery.

2. For predicates that use 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.

3. For the following predicates, a row fullselect can be used for the subquery. This means that the subquery can return more than one value for a row.
   - Basic predicate with equal or not equal comparisons
   - Quantified predicates using =ANY, =ALL, and =SOME
   - IN and NOT IN predicates

   If a row fullselect is used:
   - The select list must not contain SELECT *. Explicit values must be specified.
   - A row fullselect must be compared to a row expression. A row expression is a list of values enclosed in parentheses. There must be the same number of values returned from the subquery as there are in the row expression.
   - The row expression for an IN or NOT IN predicate cannot contain an untyped parameter marker. Use CAST to supply a result data type for these parameter markers.
   - The subquery cannot contain UNION, EXCEPT, or INTERSECT or a correlated reference.

4. A subquery cannot include the ORDER BY, FOR READ ONLY, FETCH FIRST n ROWS, UPDATE, or OPTIMIZE clause.

Including subqueries in the WHERE or HAVING clause:

You can include a subquery in a WHERE or HAVING clause by using a basic or quantified comparison, the IN keyword, or the EXISTS keyword.

Basic comparisons

You can use a subquery before or after any of the comparison operators. The subquery can return only one row. It can return multiple values for the row if the equal or not equal operators are used. SQL compares each value from the subquery row with the corresponding value on the other side of the comparison operator. For example, suppose that 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 row, a subquery can return no rows. 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 rows, 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 rows 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 of the expression must be greater than the result for each of the rows (that is, greater than the highest value) returned by the subquery. If the subquery returns an empty set (that is, no rows 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 rows 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 rows (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.

IN keyword

You can use IN to say that the value in the expression must be among the rows returned by the subquery. Using IN is equivalent to using =*ANY* or =*SOME*. Using ANY and SOME were previously described. You can also use the IN keyword with the NOT keyword in order to select rows when the value is not among the rows returned by the subquery. For example, you can 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 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. 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 change from row to row.

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 can 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 can use the following:

```
... WHERE NOT EXISTS (SELECT ...)
```

Correlated subqueries

A *correlated subquery* is a subquery that SQL might need to re-evaluate when it examines each new row (the WHERE clause) or each group of rows (the HAVING clause) in the outer-level SELECT statement.

Correlated names and references:

A correlated reference can appear in a search condition in a subquery. The reference is always in the form of 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 can 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 can be defined in B, A, or the outer-level SELECT. To define a correlation name, 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 can be defined in the outer-level SELECT, while another can 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 is B01, and the subquery for his row delivers the average education level for department B01.

The result table produced by the query has the following values.

Table 25. 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	LUCCHESSI	A00	19
000160	PIANKA	D11	17
000180	SCOUTTEN	D11	17
000210	JONES	D11	17
000220	LUTZ	D11	18
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

Table 25. Result set for previous query (continued)

EMPNO	LASTNAME	WORKDEPT	EDLEVEL
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 salaries are higher than the average salaries of their areas (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 results in the average salary for the area where department B01 belongs.

The result table produced by the query has the following values.

WORKDEPT	AVG SALARY
D21	25668.57
E01	40175.00
E21	24086.66

Example: Correlated subquery in a select-list:

Suppose that you want a list of all the departments, including the department name, number, and manager's name.

Department names and numbers are found in the CORPDATA.DEPARTMENT table. However, DEPARTMENT has only the manager's number, not the manager's name. To find the name of the manager for each department, you need 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 managers assigned are to be returned. Execute the following SQL statement:

```
SELECT DEPTNO, DEPTNAME,
    (SELECT FIRSTNME 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 has the following values.

Table 26. Result set for previous query

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

Example: Correlated subquery in an UPDATE statement:

When you use a correlated subquery in an UPDATE statement, the correlation name refers to the rows that you want to update.

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 can use the following SQL statement 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 before 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.

Example: Correlated subquery in a DELETE statement:

When you use a correlated subquery in a DELETE statement, the correlation name represents the row that you want to delete. SQL evaluates the correlated subquery once for each row in the table named in the DELETE statement to decide whether to delete the row.

Suppose that a row in the CORPDATA.PROJECT table is deleted. Rows related to the deleted project in the CORPDATA.EMPPROJACT table must also be deleted. To do this, run the following statement:

```
DELETE FROM CORPDATA.EMPPROJACT 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.

Sort sequences and normalization in SQL

A sort sequence defines how characters in a character set relate to each other when they are compared or ordered. Normalization allows you to compare strings that contain combining characters.

The sort sequence is used for all character, and UCS-2 and UTF-16 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.

Related reference

"Creating and using views" on page 31
A view can be used to access data in one or more tables or views. You create a view by using a SELECT statement.

"Creating indexes" on page 36
You can use indexes to sort and select data. In addition, indexes help the system retrieve data faster for better query performance.

"Specifying a search condition using the WHERE clause" on page 39
The WHERE clause specifies a search condition that identifies the row or rows that you want to retrieve, update, or delete.

"GROUP BY clause" on page 42
The GROUP BY clause allows you to find the characteristics of groups of rows rather than individual rows.

"HAVING clause" on page 44
The HAVING clause specifies a search condition for the groups selected by the GROUP BY clause.

"ORDER BY clause" on page 45
The ORDER BY clause specifies the particular order in which you want selected rows returned. The order is sorted by ascending or descending collating sequence of a column's or an expression's value.

"Handling duplicate rows" on page 51
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 search condition of the select-statement. Some of the rows in the result table might be duplicate.

"Defining complex search conditions" on page 52
In addition to the basic comparison predicates, such as = and >, a search condition can contain any of these predicates: BETWEEN, IN, EXISTS, IS NULL, and LIKE.

"Using the UNION keyword to combine subselects" on page 71
Using the UNION keyword, you can combine two or more subselects to form a fullselect.

Collating sequence

Sort sequence used with ORDER BY and row selection

The examples show how rows are ordered and selected for the sort sequence used.

The values in the JOB column are in mixed case. You can see the values 'Mgr', 'MGR', and 'mgr'.

Table 27. 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

Sort sequences affect ordering done by the ORDER BY clause.

The following SQL statement causes the result table to be sorted using the values in the JOB column:
```
SELECT * FROM STAFF ORDER BY JOB
```

The following table shows the result 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 28. Result of 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

The following table 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 29. Result of 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

The following table 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 this table, notice that all the values 'MGR', 'mgr' and 'Mgr' are mixed together.

Table 30. Result of 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

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
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

Sort sequence and row selection

Sort sequences affect selection of data.

The following SQL statement selects rows with the value 'MGR' in the JOB column:

```
SELECT * FROM STAFF WHERE JOB='MGR'
```

The first table shows how row selection is done with a *HEX sort sequence. 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 31. Result of using the *HEX sort sequence*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

Table 2 shows how row selection is done with a unique-weight sort sequence. The lowercase and uppercase letters are treated as unique. The lowercase 'mgr' is not treated the same as uppercase 'MGR'. Therefore, the lowercase 'mgr' is not selected.

Table 32. Result of 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

The following table shows how row selection is done with a shared-weight sort sequence. 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 all the values 'mgr', 'Mgr' and 'MGR' are selected.

Table 33. Result of 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 is in effect when the CREATE VIEW statement is 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 or UTF-16 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 is as follows:

```
CREATE VIEW V1 AS SELECT *
   FROM STAFF
   WHERE JOB = 'MGR' AND ID < 100
```

Table 34. "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 above. The query shown below is run with a sort sequence of SRTSEQ(*LANGIDUNQ) and LANGID(ENU).

Table 35. "SELECT * FROM V1 WHERE JOB = 'MGR'" using the unique-weight sort sequence for ENU language identifier

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 with the sort sequence that is in effect when the CREATE INDEX statement is 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 and UTF-16 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 or UTF-16 graphic keys do not need to be converted before comparison. This improves the performance of the query.

> **Related concepts**
>
> Using indexes with sort sequence

Sort sequence and constraints

Unique constraints are implemented with indexes. If the table on which a unique constraint is added is defined with a sort sequence, the index is created with the same sort sequence.

If defining a referential constraint, the sort sequence between the parent and dependent table must match.

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.

ICU sort sequence

When an International Components for Unicode (ICU) sort sequence table is used, the database uses the language-specific rules to determine the weight of the data based on the locale of the table.

An ICU sort sequence table named en_us (United States locale) can sort data differently than another ICU table named fr_FR (French locale) for example.

The ICU support (i5/OS option 39) properly handles data that is not normalized, producing the same results as if the data were normalized. The ICU sort sequence table can sort all character, graphic, and Unicode (UTF-8, UTF-16 and UCS-2) data.

For example, a UTF-8 character column named NAME contains the following names (the hex values of the column are given as well).

NAME	HEX (NAME)
Gómez	47C3B36D657A
Gomer	476F6D6572
Gumby	47756D6279

A *HEX sort sequence orders the NAME values as follows.

NAME
Gomer
Gumby
Gómez

An ICU sort sequence table named en_us correctly orders the NAME values as follows.

NAME
Gomer
Gómez
Gumby

When an ICU sort sequence table is specified, the performance of SQL statements that use the table can be much slower than the performance of SQL statements that use a non-ICU sort sequence table or use a *HEX sort sequence. The slower performance results from calling the ICU support to get the weighted value for each piece of data that needs to be sorted. An ICU sort sequence table can provide more sorting function but at the cost of slower running SQL statements. However, indexes created with an ICU sort sequence table can be created over columns to help reduce the need of calling the ICU support. In this case, the index key already contains the ICU weighted value, so there is no need to call the ICU support.

Related concepts

International Components for Unicode

Normalization

Normalization allows you to compare strings that contain combining characters.

Data tagged with a UTF-8 or UTF-16 CCSID can contain combining characters. Combining characters allow a resulting character to be composed of more than one character. After the first character of the compound character, one of many different non-spacing characters such as umlauts and accents can follow in the data string. If the resulting character is one that is already defined in the character set, normalization of the string results in multiple combining characters being replaced by the value of the defined character. For example, if your string contained the letter 'a' followed by an '..', the string is normalized to contain the single character 'ä'.

Normalization makes it possible to accurately compare strings. If data is not normalized, two strings that look identical on the display may not compare equal since the stored representation can be different. When UTF-8 and UTF-16 string data is not normalized, it is possible that a column in a table can have

one row with the letter 'a' followed by the umlaut character and another row with the combined 'ä' character. These two values are not both compare equal in a comparison predicate: WHERE C1 = 'ä'. For this reason, it is recommended that all string columns in a table are stored in normalized form.

You can normalize the data yourself before inserting or updating it, or you can define a column in a table to be automatically normalized by the database. To have the database perform the normalization, specify NORMALIZED as part of the column definition. This option is only allowed for columns that are tagged with a CCSID of 1208 (UTF-8) or 1200 (UTF-16). The database assumes all columns in a table have been normalized.

The NORMALIZED clause can also be specified for function and procedure parameters. If it is specified for an input parameter, the normalization will be done by the database for the parameter value before invoking the function or procedure. If it is specified for an output parameter, the clause is not enforced; it is assumed that the user's routine code will return a normalized value.

The NORMALIZE_DATA option in the QAQQINI file is used to indicate whether the system is to perform normalization when working with UTF-8 and UTF-16 data. This option controls whether the system will normalize literals, host variables, parameter markers, and expressions that combine strings before using them in SQL. The option is initialized to not perform normalization. This is the correct value for you if the data in your tables and any literal values in your applications is always normalized already through some other mechanism or never contains characters which will need to be normalized. If this is the case, you will want to avoid the overhead of system normalization in your query. If your data is not already normalized, you will want to switch the value of this option to have the system perform normalization for you.

> **Related tasks**
>
> Controlling queries dynamically with the query options file QAQQINI

Data protection

The DB2 for i5/OS database provides various methods for protecting SQL data from unauthorized users and for ensuring data integrity.

Security for SQL objects

All objects on the system, including SQL objects, are managed by the system security function.

Users can authorize SQL objects through either the SQL GRANT and REVOKE statements or the Edit Object Authority (EDTOBJAUT), Grant Object Authority (GRTOBJAUT), and Revoke Object Authority (RVKOBJAUT) CL commands.

The SQL GRANT and REVOKE statements operate on SQL functions, SQL packages, SQL procedures, distinct types, sequences, 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.

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.

> **Related reference**
>
> Security reference
>
> GRANT (Table or View Privileges)
>
> REVOKE (Table or View Privileges)

Authorization ID

An *authorization ID* is a user profile object that identifies a unique user. You can use the Create User Profile (CRTUSRPRF) command to create an authorization ID.

Views

Views 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

The DB2 for i5/OS database is designed to comply with the U.S. government C2 security level. A key feature of the C2 level is the ability to perform auditing on the system.

DB2 for i5/OS uses the audit facilities that are managed by the system security function. Auditing can be performed at an object level, a user level, or a system level. The system value QAUDCTL controls whether auditing is performed at the object or user level. The Change User Audit (CHGUSRAUD) and Change Object Audit (CHGOBJAUD) commands specify which users and objects are audited. The system value QAUDLVL controls what types of actions are audited (for example, authorization failures; and create, delete, grant, or revoke operations).

DB2 for i5/OS can also audit row changes through the DB2 for i5/OS journal support.

In some cases, entries in the auditing journal will not be in the same order as they occured. 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.

Related reference

Security reference

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).

Related reference

XA APIs

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 for i5/OS 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 for i5/OS acquires locks on rows to ensure integrity. However, some situations require DB2 for i5/OS to acquire a more exclusive table-level lock instead of row locks.

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 timeout values are supported. DB2 for i5/OS 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 data manipulation language (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 data definition language (DDL) statements and the LOCK TABLE statement is the job default wait time (DFTWAIT). You can change this value by using the CL command Change Job (CHGJOB) or Change Class (CHGCLS).

If a large record wait time is specified, deadlock detection is provided. For example, assume that 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 waits because the second job is holding the lock. If the second job then attempts to lock row 1, DB2 for i5/OS detects that the two jobs are in a deadlock and an error is 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. Using COMMIT(*RR) will also prevent other users from using a table during a unit of work.

To improve performance, DB2 for i5/OS frequently leaves the open data path (ODP) open. 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 might prevent another job from performing an operation on that table. In most cases, however, DB2 for i5/OS can detect that other jobs are holding locks and events can be signalled to those jobs. The event causes DB2 for i5/OS to close any ODPs (and release the table locks) that are associated with that table and are currently only open for performance reasons.

Note: The lock wait timeout must be large enough for the events to be signalled and the other jobs to close the ODPs, or an error is 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. Typically, 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 before 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 performs 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 does not reflect changes that are made after the cursor was opened. For information about when a temporary result table is created, see DECLARE CURSOR in the SQL reference topic collection.
- A basic subquery is evaluated when the query is opened.

You can use the SKIP LOCKED DATA clause to dramatically minimize transaction wait time. It allows a transaction to skip over rows that are incompatibly locked by other transactions without waiting for the locks to be released. This function provides flexibility and increased performance to applications that require only available and committed data to be returned. SKIP LOCKED DATA should be used only if your application does not require complete information from the query.

- The SKIP LOCKED DATA clause can be specified for a select-statement, SELECT INTO, PREPARE, searched UPDATE, or searched DELETE statement. SKIP LOCKED DATA applies to all tables referenced in the statement. If there are multiple tables involved in a query, the number of committed rows available to be returned might be reduced because rows in many tables can be locked by other transactions.
- If the isolation level for the statement is COMMIT(*NONE), COMMIT(*CHG), or COMMIT(*RR), the SKIP LOCKED DATA clause is ignored.

   Related reference

   LOCK TABLE

Journaling

The DB2 for i5/OS journal support provides 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 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 that DB2 for i5/OS created in the schema (QSQJRN). A table created in a library also has journaling started if a journal named QSQJRN 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 save time, it is journaled to the same journal at restore time. If the table was not journaled at save time, it is not journaled at restore time.

The journal created in the SQL schema 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.

Related concepts

Journal management

Related reference

"Updating tables with referential constraints" on page 88
If you are updating a parent table, you cannot modify a primary key for which dependent rows exist.

Commitment control

The DB2 for i5/OS commitment control support provides a means for processing a group of database changes, such as update, insert, or delete operations or data definition language (DDL) operations, as a single unit of work (also referred to as a *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 SCHEMA
- 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 run with an isolation level other than COMMIT(*NONE) or a RELEASE statement is run, then DB2 for i5/OS sets up the commitment control environment by implicitly calling the CL command Start Commitment Control (STRCMTCTL). DB2 for i5/OS specifies the NFYOBJ(*NONE) and CMTSCOPE(*ACTGRP) parameters, along with the LCKLVL parameter, on the STRCMTCTL command. The LCKLVL parameter specified is the lock level on the COMMIT parameter of the Create SQL (CRTSQL*xxx*), Start SQL Interactive Session (STRSQL), or Run SQL Statements (RUNSQLSTM) command. In REXX, the LCKLVL parameter specified is the lock level on the SET OPTION statement. You can use the STRCMTCTL command to specify a different CMTSCOPE, NFYOBJ, or LCKLVL parameter. If you specify CMTSCOPE(*JOB) to start the job-level commitment definition, DB2 for i5/OS uses the job-level commitment definition for programs in that activation group.

Notes:

- When commitment control is used, the tables that are referred to in the application program by data manipulation language (DML) statements must be journaled.
- The LCKLVL parameter 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 of the CRTSQL*xxx*, STRSQL, or RUNSQLSTM command will override the default lock level. Also, the LCKLVL parameter only applies to commitment control operations that are requested through the i5/OS traditional system interface (non-SQL). The lock level specified on the LCKLVL parameter is not affected by any subsequent changes to the SQL isolation level that are made by using, for example, the SET TRANSACTION statement.

For cursors that use aggregate 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 for i5/OS locks all referenced tables in shared mode (*SHRNUP). The lock prevents concurrent application processes from processing 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 might also be sent.

For cursors where 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 for i5/OS locks all referenced tables in shared mode (*SHRNUP). This prevents concurrent processes from processing anything but read-only operations on the named table or tables. A message (either SQL7902 or CPI430A) is sent that says COMMIT(*ALL) is requested but not allowed. Message SQL0595 might 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.

In a highly contentious environment using COMMIT(*RR), an application might need to retry an operation after getting an SQL0913 to allow the database unlocking mechanism time to work.

If an isolation level other then 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 for i5/OS uses locks on rows to keep other jobs from accessing changed data before a unit of work is completed. 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 is completed. This does 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.

COMMIT HOLD and ROLLBACK HOLD allow you to keep the cursor open and start another unit of work without issuing an OPEN statement again. The HOLD value is not available when you are connected to a remote database that is not on a System i platform. However, the WITH HOLD option on DECLARE CURSOR can 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 a System i platform. Such a cursor is closed on a rollback.

Table 36. 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 6) *ALL (See note 2 and 7)	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 6) *ALL (See note 2 and 7)	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 is not updated or deleted from read until next FETCH When row is updated from read until next FETCH When row is deleted from read until next DELETE When row is not updated or deleted from read until next FETCH When row is updated or deleted from read until COMMIT or ROLLBACK When row is 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	From read until next FETCH 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

Table 36. Row lock duration (continued)

SQL statement	COMMIT parameter (see note 5)	Duration of row locks	Lock type
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
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

Notes:

1. A cursor is open with UPDATE or DELETE capabilities if the result table is not read-only and if one of the following is true:
   - 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:
     – 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. 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.

7. If the USE AND KEEP EXCLUSIVE LOCKS clause is specified with the *RS or *RR isolation level, an UPDATE lock on the row will be obtained instead of a READ lock.

Related concepts

Commitment control

Related reference

DECLARE CURSOR

Isolation level

XA APIs

Savepoints

A *savepoint* is a named entity that represents the state of data and schemas at a particular point within a unit of work. You can create savepoints within a transaction. If the transaction rolls back, changes are undone to the specified savepoint, rather than to the beginning of the transaction.

You can set a savepoint 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 can 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.

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.

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 ends when:
A new unit of work is started	COMMIT or ROLLBACK is issued
A trigger is invoked	The trigger completes
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.

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
- REFRESH TABLE

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 database operation:

- ALTER FUNCTION
- ALTER PROCEDURE
- ALTER SEQUENCE
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DROP ALIAS
- DROP DISTINCT TYPE
- DROP FUNCTION
- DROP PROCEDURE
- DROP SCHEMA
- DROP SEQUENCE
- DROP TRIGGER
- RENAME (See note 1)

Note: 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 statement can be interrupted after the DB2 for i5/OS 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 might need to drop the object and then create it again. In the case of a DROP SCHEMA statement, you might need to drop the schema again or use the CL command Delete Library (DLTLIB) to remove the remaining parts of the schema.

Constraints
The DB2 for i5/OS database 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 for i5/OS enforces the validity of the constraint during any data manipulation language (DML) statement. Certain operations (such as restoring the dependent table), however, cause the validity of the constraint to be unknown. In this case, DML statements might be prevented until DB2 for i5/OS has verified the validity of the constraint.

- Unique constraints are implemented with indexes. If an index that implements a unique constraint is not valid, the Edit Rebuild of Access Paths (EDTRBDAP) command can be used to display any indexes that currently require being rebuilt.

- If DB2 for i5/OS 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 being rebuilt.

  Related concepts

  "Constraints" on page 10
  A *constraint* is a rule enforced by the database manager to limit the values that can be inserted, deleted, or updated in a table.

Adding and using check constraints:

A *check constraint* ensures the validity of data during insert and update operations by limiting the allowed 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)

fails 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 is successful:

INSERT INTO T1 **VALUES** (1, 1, 1)

Once that row is inserted, the following statement fails:

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 is not 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).

  Related reference

  ALTER TABLE
  CREATE TABLE

Save and restore functions

The i5/OS save and restore functions are used to save SQL objects to disk (save file) or to external media.

The saved versions can be restored onto any i5/OS operating system later. The save and restore functions allow an entire schema, selected objects, or only objects that have been changed since a given date and time to be saved. All information needed to restore an object to its previous state is saved. You can use the functions to recover from damage to individual tables by restoring the data with a previous version of the table or the entire schema.

When a program or service program that was created for an SQL procedure, an SQL function, or a sourced function is restored, it is automatically added to the SYSROUTINES and SYSPARMS catalogs, as long as a procedure or function does not already exist with the same signature and program name. 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.

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.

When a *DTAARA for a sequence is restored, the sequence is automatically added to the SYSSEQUENCES catalog. If the catalog is not successfully updated, the *DTAARA will be modified so it cannot be used as a sequence and an SQL9020 informational message will be output in the job log.

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.

Note: 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 DB2 for i5/OS database provides several mechanisms to reduce or eliminate damage caused by disk errors.

For example, mirroring, checksums, and Redundant Array of Independent Disks (RAID) can all reduce the possibility of disk problems. The DB2 for i5/OS 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.

If a disk error has damaged a small portion of the rows in a table, the DB2 for i5/OS database manager allows you to read rows that are still accessible.

Index recovery
The DB2 for i5/OS database provides several functions to deal with index recovery.
- System managed index protection

  The Edit Recovery for Access Paths (EDTRCYAP) CL command allows you to instruct DB2 for i5/OS 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 for i5/OS provides an index journaling function that makes it unnecessary to rebuild an entire index in the event of a power or system failure. If the index is journaled, the system database support

automatically makes sure that 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 for i5/OS.

- 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 an abnormal system failure, if indexes are not protected by one of the previously described techniques, those indexes in the process of change might need to be rebuilt by the database manager to make sure that 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 i5/OS operating system is started). All other SQL indexes are created with the *AFTIPL recovery option (this means that after the operating system is started, indexes are asynchronously rebuilt). During an IPL, the operator can see a display showing the indexes that need to be rebuilt and their recovery options. 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

To ensure that the information in the catalog is always accurate, the DB2 for i5/OS database prevents users from explicitly changing the information in the catalog and implicitly maintains the information when an SQL object described in the catalog is changed.

The integrity of the catalog is maintained whether objects in the schema are changed by SQL statements, i5/OS CL commands, System/38™ Environment CL commands, System/36 Environment functions, or any other product or utility on a System i platform. For example, you can delete a table by running an SQL DROP statement, issuing an i5/OS Delete File (DLTF) CL command, issuing a System/38 Delete File (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 removes the description of the table from the catalog when the table is deleted. The following table lists various functions and their associated effects on the catalog.

Table 37. Effects 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

You can create a schema in a user auxiliary storage pool (ASP) using the ASP clause on the CREATE SCHEMA statement.

The Create Library (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.

> **Related concepts**
>
> Backup and recovery
>
> Disk management

Independent auxiliary storage pool

Independent disk pools are used to set up user databases on the system.

There are three types of independent disk pools: primary, secondary, and user-defined file system (UDFS). Primary independent disk pools are used to set up databases.

You can work with multiple databases. The system 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 system through the use of independent disk pools, which are set up in the disk management function of System i Navigator. After an independent disk pool is set up, it appears as another database in the Databases folder of System i Navigator.

> **Related concepts**
>
> Disk management

Routines

Routines are pieces of code or programs that you can call to perform operations.

Stored procedures

A *procedure* (often called a stored procedure) is a program that can be called to perform operations. A procedure can include both host language statements and SQL statements. Procedures in SQL provide the same benefits as procedures in a host language.

DB2 stored procedure support provides a way for an SQL application to define and then call a procedure through SQL statements. Stored procedures can be used in both distributed and nondistributed DB2 applications. One of the advantages of using stored procedures is that for distributed applications, the processing 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.

The following sections describe the SQL statements used to define and call the stored procedure, information about passing parameters to the stored procedure, and examples of stored procedure usage.

For more information about stored procedures, see Stored Procedures, Triggers, and User-Defined Functions on DB2® Universal Database™ for iSeries®

Related concepts

"Stored procedures" on page 11
A *stored procedure* is a program that can be called with the SQL CALL statement.

Java SQL routines

Related reference

"DRDA stored procedure considerations" on page 277
The i5/OS Distributed Relational Database Architecture (DRDA) server supports the return of one or more result sets in a stored procedure.

CREATE PROCEDURE

Defining an external procedure

The CREATE PROCEDURE statement for an external procedure names the procedure, defines the parameters and their attributes, and provides other information about the procedure that 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 called 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 will 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 called on the CALL statement. For example, the NULL indicator argument cannot be passed.

Related reference

"Using the embedded CALL statement where no procedure definition exists" on page 129
A static CALL statement without a corresponding CREATE PROCEDURE statement is processed with these rules.

Defining an SQL procedure

The CREATE PROCEDURE statement for an SQL procedure names the procedure, defines the parameters and their attributes, provides other information about the procedure that is used when the procedure is called, and 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 by using SQL control statements. SQL control statements consist of the following statements:

- 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 called sets variable END_TABLE to 1. This handler is called when the FETCH has no more rows to return. When the handler is called, SQLCODE and SQLSTATE are reinitialized to 0.
  - Declares an exit handler for SQLEXCEPTION. If called, DEPT_SALARY is set to NULL and the processing of the compound statement is terminated. This handler is called if any errors occur, that is, 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 called, 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.

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 FIRSTNME, MIDINIT, LASTNAME, BONUS
             FROM CORPDATA.EMPLOYEE
             WHERE WORKDEPT = CREATE_BONUS_TABLE.DEPT_NUMBER
         DO
         IF BONUS > 0 THEN
           SET NAME = FIRSTNME 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 called 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 called and decrements CNT. Processing continues on the SET statement following the INSERT statement.
  - Declares an UNDO handler for SQLEXCEPTION. If called, 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 FIRSTNME, 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,
         FIRSTNME 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, FIRSTNME, 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 fails. 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'.

Calling a stored procedure

The SQL CALL statement calls 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 typically 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.

The following types of CALL statements need to be addressed because the DB2 for i5/OS database has different rules for each type:
- 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

Notes:

> 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.

Using the 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 issued, a call to program MYLIB/PROC1 is made and two arguments are passed. Because the language of the program is ILE C, the first argument is a C NUL-terminated string, 11 characters long, which contains the contents of host variable HV1. On a call to an ILE C procedure, SQL 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 because 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 the embedded CALL statement where no procedure definition exists:

A static CALL statement without a corresponding CREATE PROCEDURE statement is processed with these 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

The following PL/I example shows an embedded CALL statement where no procedure definition exists:

```
DCL HV2 CHAR(10);
  :
EXEC SQL CALL P2 (:HV2);
  :
```

When the CALL statement is issued, SQL attempts to find the program based on standard SQL naming conventions. For the preceding example, assume that the naming option of *SYS (system naming) is used and that a DFTRDBCOL parameter is not specified on the Create SQL PL/I Program (CRTSQLPLI) command. In this case, the library list is searched for a program named P2. Because 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.

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 of 5 and a scale of 2.
- All floating point constants are passed as double-precision floating point.

Special registers specified on a dynamic CALL statement are passed as their defined data type and length with the following exceptions:

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.

Using the embedded CALL statement with an SQLDA:

In either type of embedded CALL statement (where a procedure definition might or might not exist), an SQLDA rather than a parameter list can be passed.

The following C examples illustrates this. 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.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

```
#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 sqlda* 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, 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.

Using the dynamic CALL statement where no CREATE PROCEDURE exists:

These 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

The following C example shows 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 is 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.

Examples: CALL statements:

These examples show how the arguments of a CALL statement are passed to a procedure for several languages, and how the arguments are received into local variables in the procedure.

Example 1: ILE C and PL/I procedures called from an ILE C program:

This example shows an ILE C program that uses the CREATE PROCEDURE definitions to call the P1 and P2 procedures. Procedure P1 is written in ILE C and has 10 parameters. Procedure P2 is written in PL/I and also has 10 parameters.

Defining the P1 and P2 procedures

```
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
```

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

Calling the P1 and P2 procedures

```
/****************************************************************/
/*********** 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 */

/*****************************************************/
/* 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 before 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 *********************************/
/**************************************************************/
```

Procedure P1

```
/******** 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 can 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];
/*********************************************************/
/* 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 works 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 *****************************/
```

Procedure P2

```
/******** 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;
```

Example 2: A REXX procedure called from an ILE C program:

This example shows a REXX procedure called from an ILE C program.

Defining the REXX procedure

```
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
```

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

Calling the REXX procedure

```
/**********************************************************/
/*********** START OF SQL C Application ******************/

        #include <decimal.h>
        #include <stdio.h>
        #include <string.h>
        #include <wcstr.h>
        /*-------------------------------------------------------*/
        exec sql include sqlca;
        exec sql include sqlda;
        /* ******************************************************/
        /* 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()
      {
      /* ***********************************************************/
      /* 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    ***********************************/
/**********************************************************************/

/**********************************************************************/
/****** 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'SSTTUUVVWWXXYYZZAA'" 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

/***********************************************************/
/* 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   **********************************/
```

Returning result sets from stored procedures

In addition to returning output parameters, a stored procedure can return a result set (that is, a result table associated with a cursor opened in the stored procedure) to the application that issues the CALL statement. The application can then issue fetch requests to read the rows of the result set cursor.

Whether a result set gets returned depends on the returnability attribute of the cursor. The cursor's returnability attribute can be explicitly given in the DECLARE CURSOR statement or it can be defaulted. The SET RESULT SETS statement also allows for an indication of where the result sets should be returned. By default, cursors which are opened in a stored procedure are defined to have a returnability attribute of RETURN TO CALLER. To return the result set associated with the cursor to the application which called the outermost procedure in the call stack, the returnability attribute of RETURN TO CLIENT is specified on the DECLARE CURSOR statement. This will allow inner procedures to return result sets when the application calls nested procedures. For cursors whose result sets are never to be returned to caller or client, the returnability attribute of WITHOUT RETURN is specified on the DECLARE CURSOR statement.

Note: When you use COBOL, the result sets are automatically closed because of the setup of the COBOL program. Change the EXIT PROGRAM statement to EXIT PROGRAM AND CONTINUE RUN UNIT and the result sets should be returned.

There are many cases where opening the cursor in a stored procedure and returning its result set provides advantages over opening the cursor directly in the application. For instance, security to the tables referenced in the query can be adopted from the stored procedure so that users of the application do not need to be granted direct authority to the tables. Instead, they are given authority to call the stored procedure, which is compiled with adequate authority to access the tables. Another advantage to opening the cursors in the stored procedure is that multiple result sets can be returned from a single call to the stored procedure, which can be more efficient that opening the cursors separately from the calling application. Additionally, each call to the same stored procedure may return a different number of result sets, providing some application versatility.

The interfaces that can work with stored procedure result sets include JDBC, CLI, and ODBC. An example of how to use these API interfaces for working with stored procedure result sets is included in the following examples.

Example 1: Calling a stored procedure that returns a single result set:

This example shows the API calls that an Open Database Connectivity (ODBC) application can use to call a stored procedure to return a result set.

Note that in this example the DECLARE CURSOR statement does not have an explicit returnability specified. When there is only a single stored procedure on the call stack, the returnability attribute of RETURN TO CALLER as well as that of RETURN TO CLIENT will make the result set available to the caller of the application. Also note that the stored procedure is defined with a DYNAMIC RESULT SETS clause. For SQL procedures, this clause is required if the stored procedure will be returning result sets.

Defining the stored procedure:

```
PROCEDURE prod.resset

CREATE PROCEDURE prod.resset () LANGUAGE SQL
 DYNAMIC RESULT SETS 1
 BEGIN
 DECLARE C1 CURSOR FOR SELECT * FROM QIWS.QCUSTCDT;
 OPEN C1;
 RETURN;
 END
```

ODBC application

Note: Some of the logic has been removed.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

```
   :
strcpy(stmt,"call prod.resset()");
rc = SQLExecDirect(hstmt,stmt,SQL_NTS);
if (rc == SQL_SUCCESS)
 {
   // CALL statement has executed successfully. Process the result set.
   // Get number of result columns for the result set.
   rc = SQLNumResultCols(hstmt, &wNum);
   if (rc == SQL_SUCCESS)
      // Get description of result columns in result set
     { rc = SQLDescribeCol(hstmt,à);
       if (rc == SQL_SUCCESS)
           :
```

```
  {
   // Bind result columns based on attributes returned
   //
     rc = SQLBindCol(hstmt,à);
    :
   // FETCH records until EOF is returned

    rc = SQLFetch(hstmt);
    while (rc == SQL_SUCCESS)
      { // process result returned on the SQLFetch
            :
        rc = SQLFetch(hstmt);
      }
     :
     }
   // Close the result set cursor when done with it.
   rc = SQLFreeStmt(hstmt,SQL_CLOSE);
    :
```

Example 2: Calling a stored procedure that returns a result set from a nested procedure:

This example shows how a nested stored procedure can open and return a result set to the outermost procedure.

To return a result set to the outermost procedure in an environment where there are nested stored procedures, the RETURN TO CLIENT returnability attribute should be used on the DECLARE CURSOR statement or on the SET RESULT SETS statement to indicate that the cursors are to be returned to the application which called the outermost procedure. Note that this nested procedure returns two result sets to the client; the first, an array result set, and the second a cursor result set. Both an ODBC and a JDBC client application are shown below along with the stored procedures.

Defining the stored procedures

CREATE PROCEDURE prod.rtnnested () **LANGUAGE CL DYNAMIC RESULT SET 2**
       EXTERNAL NAME prod.rtnnested **GENERAL**

CREATE PROCEDURE prod.rtnclient () **LANGUAGE RPGLE**
       EXTERNAL NAME prod.rtnclient **GENERAL**

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

CL source for stored procedure prod.rtnnested
```
PGM
            CALL      PGM(PROD/RTNCLIENT)
```

ILE RPG source for stored procedure prod.rtnclient
```
DRESULT           DS                    OCCURS(20)
D COL1                     1    16A
C    1              DO        10        X                    2 0
C    X              OCCUR     RESULT
C                   EVAL      COL1='array result set'
C                   ENDDO
C                   EVAL      X=X-1
C/EXEC SQL DECLARE C2 CURSOR WITH RETURN TO CLIENT
C+ FOR SELECT LSTNAM FROM QIWS.QCUSTCDT FOR FETCH ONLY
C/END-EXEC
C/EXEC SQL
C+  OPEN C2
C/END-EXEC
C/EXEC SQL
C+ SET RESULT SETS FOR RETURN TO CLIENT ARRAY :RESULT FOR :X ROWS,
```

```
C+  CURSOR C2
C/END-EXEC
C                      SETON                                            LR
C                      RETURN
```

ODBC application

```
//*******************************************************************
//
// Module:
//    Examples.C
//
// Purpose:
//    Perform calls to stored procedures to get back result sets.

//
// *******************************************************************


#include "common.h"
#include "stdio.h"



// *******************************************************************
//
// Local function prototypes.
//
// *******************************************************************

SWORD   FAR PASCAL RetClient(lpSERVERINFO lpSI);
BOOL    FAR PASCAL Bind_Params(HSTMT);
BOOL    FAR PASCAL Bind_First_RS(HSTMT);
BOOL    FAR PASCAL Bind_Second_RS(HSTMT);


// *******************************************************************
//
// Constant strings definitions for SQL statements used in
// the auto test.
//
// *******************************************************************
//
// Declarations of variables global to the auto test.
//
// *******************************************************************
#define ARRAYCOL_LEN 16
#define LSTNAM_LEN 8
char    stmt[2048];
char    buf[2000];

UDWORD rowcnt;
char arraycol[ARRAYCOL_LEN+1];
char lstnam[LSTNAM_LEN+1];
SDWORD cbcol1,cbcol2;

lpSERVERINFO lpSI;  /* Pointer to a SERVERINFO structure. */



// *******************************************************************
//
//  Define the auto test name and the number of test cases
//      for the current auto test.  These informations will
//      be returned by AutoTestName().
//
// *******************************************************************

LPSTR szAutoTestName =  CREATE_NAME("Result Sets Examples");
UINT  iNumOfTestCases = 1;
```

```
// ********************************************************************
//
//   Define the structure for test case names, descriptions,
//      and function names for the current auto test.
//      Test case names and descriptions will be returned by
//      AutoTestDesc().  Functions will be run by
//      AutoTestFunc() if the bits for the corresponding test cases
//      are set in the rglMask member of the SERVERINFO
//      structure.
//
// ********************************************************************
struct TestCase  TestCasesInfo[] =
{
        "Return to Client",
        "2 result sets    ",
         RetClient

};


// ********************************************************************
//
// Sample return to Client:
//      Return to Client result sets. Call a CL program which in turn
//      calls an RPG program which returns 2 result sets. The first
//      result set is an array result set and the second is a cursor
//      result set.
//
//
// ********************************************************************
SWORD FAR PASCAL RetClient(lpSERVERINFO lpSI)
{
    SWORD       sRC = SUCCESS;
    RETCODE     returncode;
    HENV        henv;
    HDBC        hdbc;
    HSTMT       hstmt;



    if (FullConnect(lpSI, &henv, &hdbc, &hstmt) == FALSE)
    {
       sRC = FAIL;
       goto ExitNoDisconnect;
    }
    // *******************************************************
    // Call CL program PROD.RTNNESTED, which in turn calls RPG
    // program RTNCLIENT.
    // *******************************************************
    strcpy(stmt,"CALL PROD.RTNNESTED()");
    // ***********************************************************
    // Call the CL program prod.rtnnested. This program will in turn
    // call the RPG program proc.rtnclient, which will open 2 result
    // sets for return to this ODBC application.
    // ***********************************************************
    returncode = SQLExecDirect(hstmt,stmt,SQL_NTS);
    if (returncode != SQL_SUCCESS)
    {
       vWrite(lpSI, "CALL PROD.RTNNESTED is not Successful", TRUE);
    }
    else
    {
       vWrite(lpSI, "CALL PROC.RTNNESTED was Successful", TRUE);
    }
```

```
// ****************************************************************
// Bind the array result set output column. Note that the result
// sets are returned to the application in the order that they
// are specified on the SET RESULT SETS statement.
// ****************************************************************
if (Bind_First_RS(hstmt) == FALSE)
{
   myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
              returncode, "Bind_First_RS");
   sRC = FAIL;
   goto ErrorRet;
}
else
{
   vWrite(lpSI, "Bind_First_RS Complete...", TRUE);
}
// ****************************************************************
// Fetch the rows from the array result set. After the last row
// is read, a returncode of SQL_NO_DATA_FOUND will be returned to
// the application on the SQLFetch request.
// ****************************************************************
returncode = SQLFetch(hstmt);
while(returncode == SQL_SUCCESS)
{
 wsprintf(stmt,"array column = %s",arraycol);
 vWrite(lpSI,stmt,TRUE);
 returncode = SQLFetch(hstmt);
}
if  (returncode == SQL_NO_DATA_FOUND) ;
else {
      myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS_WITH_INFO,
                 returncode, "SQLFetch");
      sRC = FAIL;
      goto ErrorRet;
    }
// *******************************************************
// Get any remaining result sets from the call. The next
// result set corresponds to cursor C2 opened in the RPG
// Program.
// *******************************************************
returncode = SQLMoreResults(hstmt);
if (returncode != SQL_SUCCESS)
 {
  myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS, returncode, "SQLMoreResults");
  sRC = FAIL;
  goto ErrorRet;
 }
// ****************************************************************
// Bind the cursor result set output column. Note that the result
// sets are returned to the application in the order that they
// are specified on the SET RESULT SETS statement.
// ****************************************************************

if (Bind_Second_RS(hstmt) == FALSE)
{
   myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
              returncode, "Bind_Second_RS");
   sRC = FAIL;
   goto ErrorRet;
}
else
{
   vWrite(lpSI, "Bind_Second_RS Complete...", TRUE);
}
// ****************************************************************
// Fetch the rows from the cursor result set. After the last row
// is read, a returncode of SQL_NO_DATA_FOUND will be returned to
```

```
   // the application on the SQLFetch request.
   // ***************************************************************
    returncode = SQLFetch(hstmt);
   while(returncode == SQL_SUCCESS)
   {
    wsprintf(stmt,"lstnam = %s",lstnam);
    vWrite(lpSI,stmt,TRUE);
    returncode = SQLFetch(hstmt);
   }
   if (returncode == SQL_NO_DATA_FOUND) ;
   else {
         myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS_WITH_INFO,
                     returncode, "SQLFetch");
         sRC = FAIL;
         goto ErrorRet;
       }

   returncode = SQLFreeStmt(hstmt,SQL_CLOSE);
   if (returncode != SQL_SUCCESS)
   {
      myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
                        returncode, "Close statement");
      sRC = FAIL;
      goto ErrorRet;
   }
   else
   {
      vWrite(lpSI, "Close statement...", TRUE);
   }



ErrorRet:
   FullDisconnect(lpSI, henv, hdbc, hstmt);
   if (sRC == FAIL)
   {
      // a failure in an ODBC function that prevents completion of the
      // test - for example, connect to the server
      vWrite(lpSI, "\t\t *** Unrecoverable RTNClient Test FAILURE ***",  TRUE);
   } /* endif */

ExitNoDisconnect:

   return(sRC);
} // RetClient



BOOL FAR PASCAL Bind_First_RS(HSTMT hstmt)
{
 RETCODE rc = SQL_SUCCESS;
   rc = SQLBindCol(hstmt,1,SQL_C_CHAR,arraycol,ARRAYCOL_LEN+1, &cbcol1);
   if (rc != SQL_SUCCESS) return FALSE;
   return TRUE;
}
BOOL FAR PASCAL Bind_Second_RS(HSTMT hstmt)
{
 RETCODE rc = SQL_SUCCESS;
   rc = SQLBindCol(hstmt,1,SQL_C_CHAR,lstnam,LSTNAM_LEN+1,&dbcol2);
   if (rc != SQL_SUCCESS) return FALSE;
   return TRUE;
}
```

JDBC application

```
//-----------------------------------------------------------
// Call Nested procedures which return result sets to the
// client, in this case a JDBC client.
//-----------------------------------------------------------
import java.sql.*;
public class callNested
{
  public static void main (String argv[])                // Main entry point
  {
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }

    try {
      Connection jdbcCon =
DriverManager.getConnection("jdbc:db2:lp066ab","Userid","xxxxxxx");
      jdbcCon.setAutoCommit(false);
      CallableStatement cs = jdbcCon.prepareCall("CALL PROD.RTNNESTED");
      cs.execute();
      ResultSet rs1 = cs.getResultSet();
      int r = 0;
while (rs1.next())
      {
      r++;
      String s1 = rs1.getString(1);
      System.out.print("Result set 1 Row: " + r + ": ");
      System.out.print(s1 + "    " );
      System.out.println();
      }
      cs.getMoreResults();
      r = 0;
      ResultSet rs2 = cs.getResultSet();
      while (rs2.next())
      {
      r++;
      String s2 = rs2.getString(1);
      System.out.print("Result set 2 Row: " + r + ": ");
      System.out.print(s2 + "    ");
      System.out.println();
      }
  }
  catch ( SQLException e ) {
      System.out.println( "SQLState: " + e.getSQLState() );
      System.out.println( "Message : " + e.getMessage() );
      e.printStackTrace();
  }
  }    // main
}
```

Parameter passing conventions for stored procedures and user-defined functions

The CALL statement and a function call 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, as shown in the following tables. The SQL data type is contained in the leftmost column of each 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 is blank, the data type is not supported as a parameter type for that language. A host variable declaration indicates that DB2 for i5/OS 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 or function. When an SQL procedure or function is called, all SQL data types are supported so no column is provided in the table.

Table 38. Data types of parameters

SQL data type	C and C++	CL	COBOL and ILE COBOL
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.
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.
DECFLOAT	_Decimal32, _Decimal64, _Decimal128 **Note:** Only supported for C.		
REAL or FLOAT(p)	float		COMP-1 **Note:** Only supported for ILE COBOL.
DOUBLE PRECISION or FLOAT or FLOAT(p)	double		COMP-2 **Note:** Only supported for ILE COBOL.
CHARACTER(n)	char ... [n+1]	TYPE(*CHAR) LEN(n)	PIC X(n)
VARCHAR(n)	char ... [n+1]		Varying-Length Character String
VARCHAR(n) FOR BIT DATA	VARCHAR structured form		Varying-Length Character String
CLOB	CLOB structured form		CLOB structured form **Note:** Only supported for ILE COBOL.
GRAPHIC(n)	wchar_t ... [n+1]		PIC G(n) DISPLAY-1 or PIC N(n) **Note:** Only supported for ILE COBOL.
VARGRAPHIC(n)	VARGRAPHIC structured form		Varying-Length Graphic String **Note:** Only supported for ILE COBOL.

Table 38. Data types of parameters (continued)

SQL data type	C and C++	CL	COBOL and ILE COBOL
DBCLOB	DBCLOB structured form		DBCLOB structured form **Note:** Only supported for ILE COBOL.
BINARY	BINARY structured form		BINARY structured form
VARBINARY	VARBINARY structured form		VARBINARY structured form
BLOB	BLOB structured form		BLOB structured form **Note:** Only supported for ILE COBOL.
DATE	char ... [11]	TYPE(*CHAR) LEN(10)	PIC X(10) **Note:** For ILE COBOL only, FORMAT DATE.
TIME	char ... [9]	TYPE(*CHAR) LEN(8)	PIC X(8) **Note:** For ILE COBOL only, FORMAT TIME.
TIMESTAMP	char ... [27]	TYPE(*CHAR) LEN(26)	PIC X(26) **Note:** For ILE COBOL only, FORMAT TIMESTAMP.
ROWID	ROWID structured form		ROWID structured form
DataLink			
Indicator variable	short		PIC S9(4) BINARY

Table 39. Data types of parameters

SQL data type	Java™ parameter style JAVA	Java parameter style DB2GENERAL	PL/I
SMALLINT	short	short	FIXED BIN(15)
INTEGER	int	int	FIXED BIN(31)
BIGINT	long	long	
DECIMAL(p,s)	BigDecimal	BigDecimal	FIXED DEC(p,s)
NUMERIC(p,s)	BigDecimal	BigDecimal	
DECFLOAT			
REAL or FLOAT(p)	float	float	FLOAT BIN(p)
DOUBLE PRECISION or FLOAT or FLOAT(p)	double	double	FLOAT BIN(p)
CHARACTER(n)	String	String	CHAR(n)
VARCHAR(n)	String	String	CHAR(n) VAR
VARCHAR(n) FOR BIT DATA	byte[ ]	com.ibm.db2.app.Blob	CHAR(n) VAR
CLOB	java.sql.Clob	com.ibm.db2.app.Clob	CLOB structured form
GRAPHIC(n)	String	String	
VARGRAPHIC(n)	String	String	

Table 39. Data types of parameters (continued)

SQL data type	Java™ parameter style JAVA	Java parameter style DB2GENERAL	PL/I
DBCLOB	java.sql.Clob	com.ibm.db2.app.Clob	DBCLOB structured form
BINARY	byte[ ]	com.ibm.db2.app.Blob	BINARY structured form
VARBINARY	byte[ ]	com.ibm.db2.app.Blob	VARBINARY structured form
BLOB	java.sql.Blob	com.ibm.db2.app.Blob	BLOB structured form
DATE	Date	String	CHAR(10)
TIME	Time	String	CHAR(8)
TIMESTAMP	Timestamp	String	CHAR(26)
ROWID	byte[]	com.ibm.db2.app.Blob	ROWID structured form
DataLink			
Indicator variable			FIXED BIN(15)

Table 40. Data types of parameters

SQL data type	REXX	RPG	ILE RPG
SMALLINT		Data structure that contains a single sub-field. *B* in position 43, length must be 2, and 0 in position 52 of the sub-field specification.	Data specification. *B* in position 40, length must be <= 4, and 00 in positions 41-42 of the sub-field specification. or Data specification. *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. *B* in position 43, length must be 4, and 0 in position 52 of the sub-field specification.	Data specification. *B* in position 40, length must be <=09 and >=05, and 00 in positions 41-42 of the sub-field specification. or Data specification. *I* in position 40, length must be 10, and 00 in positions 41-42 of the sub-field specification.
BIGINT			Data specification. *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. *P* 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. *P* in position 40 and 00 through 31 in positions 41-42 of the sub-field specification.

Table 40. Data types of parameters (continued)

SQL data type	REXX	RPG	ILE RPG
NUMERIC(p,s)		Data structure that contains a single sub-field. *Blank* in position 43 and 0 through 9 in position 52 of the sub-field specification.	Data specification. *S* in position 40, or *Blank* in position 40 and 00 through 31 in position 41-42 of the sub-field specification.
DECFLOAT			
REAL or FLOAT(p)	String with digits, then an E, (then an optional sign), then digits		Data specification. *F* 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. *F* 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. *Blank* in position 43 and 52 of the sub-field specification. or A character input field or calculation result field.	Data specification. *A* in position 40, or *Blank* in position 40 and 41-42 of the sub-field specification.
VARCHAR(n)	String with n characters within two apostrophes		Data specification. *A* in position 40, or *Blank* 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. *A* in position 40, or *Blank* in position 40 and 41-42 of the sub-field specification and the keyword VARYING in positions 44-80.
CLOB			CLOB structured form
GRAPHIC(n)	String starting with G', then n double-byte characters, then '		Data specification. *G* in position 40 of the sub-field specification.
VARGRAPHIC(n)	String starting with G', then n double-byte characters, then '		Data specification. *G* in position 40 of the sub-field specification and the keyword VARYING in positions 44-80.
DBCLOB			DBCLOB structured form
BINARY			BINARY structured form
VARBINARY			VARBINARY structured form
BLOB			BLOB structured form
DATE	String with 10 characters within two apostrophes	Data structure field without sub-fields or data structure that contains a single sub-field. *Blank* in position 43 and 52 of the sub-field specification. Length is 10. or A character input field or calculation result field.	Data specification. *D* in position 40 of the sub-field specification. DATFMT(*ISO) in position 44-80.

Table 40. Data types of parameters (continued)

SQL data type	REXX	RPG	ILE RPG
TIME	String with 8 characters within two apostrophes	Data structure field without sub-fields or data structure that contains a single sub-field. *Blank* in position 43 and 52 of the sub-field specification. Length is 8. or A character input field or calculation result field.	Data specification. *T* in position 40 of the sub-field specification. TIMFMT(*ISO) in position 44-80.
TIMESTAMP	String with 26 characters within two apostrophes	Data structure field without sub-fields or data structure that contains a single sub-field. *Blank* in position 43 and 52 of the sub-field specification. Length is 26. or A character input field or calculation result field.	Data specification. *Z* in position 40 of the sub-field specification.
ROWID			ROWID structured form
DataLink			
Indicator variable	Numeric string with no decimal (and an optional leading sign).	Data structure that contains a single sub-field. *B* in position 43, length must be 2, and 0 in position 52 of the sub-field specification.	Data specification. *B* in position 40, length must be <=4, and 00 in positions 41-42 of the sub-field specification.

Related concepts

Embedded SQL programming

Java SQL routines

Indicator variables and stored procedures

Host variables with indicator variables can be used with the CALL statement to pass additional information to and from a procedure.

To indicate that the associated host variable contains the null value, the indicator variable is set to a negative value of -1, -2, -3, -4, or -6. A CALL statement with indicator variables is processed as follows:

- If the indicator variable is negative, a default value is passed for the associated host variable on the CALL statement and the indicator variable is passed unchanged.

- If the indicator variable is not negative, the host variable and the indicator variable are passed unchanged.

When an SQL procedure or an external procedure that was compiled without the *EXTIND option is called, the extended indicator values of -5 and -7 cannot be passed. An error is then issued on the CALL statement. When an external procedure that was compiled with the *EXTIND option is called, the extended indicator values can be passed.

The processing rules are the same for input parameters to the procedure as well as output parameters returned from the procedure. When indicator variables are used, the correct coding method 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 how the indicator variables are passed into procedure PROC1 (as a third argument that consists of an array of 2-byte values).

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

Assume that a procedure is defined as follows. The ILE RPG program was compiled to not allow extended indicators.

```
  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                 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*                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
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
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*                CODE FOR INOUTP DOES NOT CONTAIN MEANINGFUL DATA
    C*                  :
    C                 ELSE
    C*                  :
    C*                CODE FOR INOUTP CONTAINS MEANINGFUL DATA
    C*                  :
```

```
C                 ENDIF
C*                PROCESS ALL REMAINING VARIABLES
C*
C*                BEFORE RETURNING, SET OUTPUT VALUE FOR FIRST
C*                PARAMETER AND SET THE INDICATOR TO A NON-NEGATIVE
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) = -1
C                 RETURN
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
End of PROGRAM PROC1
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Returning a completion status to the calling program

SQL and external procedures return status information to the calling program in different ways.

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.

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 before 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 calls the procedure. The calling program that issues 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.

> **Related reference**
>
> SQL control statements

Using user-defined functions

In writing SQL applications, you can implement some actions or operations as a user-defined function (UDF) or as a subroutine in your application. Although it might appear easier to implement new operations as subroutines, you might want to consider the advantages of using a UDF instead.

For example, if the new operation is something that other users or programs can take advantage of, a UDF can help to reuse it. In addition, the function can be called directly in SQL wherever an expression can be used. 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.

In certain cases, calling 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 can issue the following select statement:

    SELECT A, B, C FROM T

When it receives each row, it runs the program's SELECTION_CRITERIA function 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 one 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 you deal with large objects (LOBs). Suppose that you have a function that extracts some information from a value of a LOB. 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 can be enormous, depending on the particular situation.

Related concepts

"User-defined functions" on page 11

A *user-defined function* is a program that can be called like any built-in functions.

UDF concepts

A *user-defined function* (UDF) is a function that is defined to the DB2 database system through the CREATE FUNCTION statement and that can be referenced in SQL statements. A UDF can be an external function or an SQL function.

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* function, an *aggregate* function, or a *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), written in SQL, or sourced (using the implementation of an existing function).

An *aggregate function* receives a set of like values (a column of data) and returns a single value answer from this set of values. Some built-in functions are aggregate functions. An example of an aggregate function is the built-in function AVG(). An external UDF cannot be defined as an aggregate function. However, a sourced UDF is defined to be an aggregate function if it is sourced on one of the built-in aggregate functions. The latter is useful for distinct types. For example, if a distinct type SHOESIZE exists that is defined with base type INTEGER, you can define a UDF, AVG(SHOESIZE), as an aggregate function sourced on the existing built-in aggregate 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 can, 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.

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. The path is an ordered list of schema names that is used for resolving unqualified references to UDFs and UDTs. In cases where a function reference matches a function 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 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 is the qualified function name and the data types of all the function parameters in the order in that they are defined.

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 requires an scalar function, or vice versa.

Length of time that the UDF runs

UDFs are called 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 called from a low level of the database.

As a consequence of being called from such a low level, there are certain resources (locks and seizes) being held at the time the UDF is called 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 calling 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 calling the UDF will fail.

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 want to increase the wait time, this can be done using the UDF_TIME_OUT option in the query 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.

Related tasks

Controlling queries dynamically with the query options file QAQQINI

Writing UDFs as SQL functions

An *SQL function* is a user-defined function (UDF) that you define, write, and register using the CREATE FUNCTION statement.

An SQL function is written only in the SQL language and its 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 passed.

Example: SQL scalar UDFs:

This example shows a scalar 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 can then be called as:

```
SELECT ORDERNBR, PRIORITY(ORDERDUEDATE) FROM ORDERS
```

Example: SQL table UDFs:

This example shows a table 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 PROJACT
     WHERE ACSTDATE<=indate;
  END
```

The function can then be called 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 write the executable code of a user-defined function (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.

> **Related concepts**
>
> Java SQL routines

Registering UDFs:

A user-defined function (UDF) must be registered in the database before the function can be recognized and used by SQL. 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 after you have written and completely tested the actual code. It is possible to define the UDF before actually writing it. However, to avoid any problems with running your UDF, you are encouraged to write and test it extensively before registering it.

> **Related reference**
>
> CREATE FUNCTION

Example: Exponentiation:

Suppose that you write an external function to perform exponentiation of floating point values, and you want 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 RETURNS NULL ON NULL INPUT is specified since you want the result to be NULL if either argument is NULL. As there is no reason why EXPON cannot be parallel, the ALLOW PARALLEL value is specified.

Example: String search:

Suppose that you write a user-defined function (UDF) to look for a given string, passed as an argument, within a given character large object (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.

The C program was written 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 FINDSTRING
  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 program really returns a FLOAT value, but you want to cast this to INTEGER before returning the value to the SQL statement which used the UDF. Also, you want to provide your own specific name for the function. Because the UDF was not written to handle NULL values, you use the RETURNS NULL ON NULL INPUT.

Example: BLOB string search:

Suppose that you want the FINDSTRING function to work on binary large objects (BLOBs) as well as on character large objects (CLOBs). To do this, you define another FINDSTRING function that takes BLOB as the first parameter.

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC FINDSTRING_BLOB
  EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  RETURNS NULL ON NULL INPUT
```

This example illustrates overloading of the UDF name and shows that multiple UDFs can share the same program. 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 interface for BLOB and CLOB between DB2 and the UDF program is the same: length followed by data.

Example: String search over a user-defined type (UDT):

Suppose that you are satisfied with the FINDSTRING function from the binary large object (BLOB) string search, but now you want to define 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. Note the further overloading of FINDSTRING in this example:

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INT
  SPECIFIC "slick_fboat"
  SOURCE SPECIFIC FINDSTRING_BLOB
```

Note that this FINDSTRING function has a different signature from the FINDSTRING functions in "Example: BLOB string search" on page 157, so there is no problem overloading the name. 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 157. 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: AVG over a user-defined type (UDT):

This example implements the AVG aggregate function over the CANADIAN_DOLLAR distinct type.

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.

```
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. By definition, it is a NOT DETERMINISTIC function because its answer varies from call to call.

It uses the SCRATCHPAD to save the last value returned. Each time it is called, 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:

Suppose that you write a table function that returns a row consisting of a single document identifier column for each known document that matches a given subject area (the first parameter) and contains the given string (second parameter).

This user-defined function (UDF) quickly identifies 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. The RETURNS clause defines the output from DOCMATCH, including the column name DOC_ID. FINAL CALL does not need to be specified for this table function. The DISALLOW PARALLEL keyword is required since table functions cannot operate in parallel. Although the size of the output from DOCMATCH can be a large table, CARDINALITY 20 is a representative value, and is specified to help the optimizer make good decisions.

Typically, this table function is 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.

Passing arguments from DB2 to external functions:

DB2 provides storage for all parameters that are passed to a user-defined function (UDF). Therefore, parameters are passed to an 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 SQL parameter style conforms to the industry standard SQL. This parameter style can be used only with scalar user-defined functions (UDFs).

With parameter style SQL, the parameters are passed into the external program as follows (in the order specified):
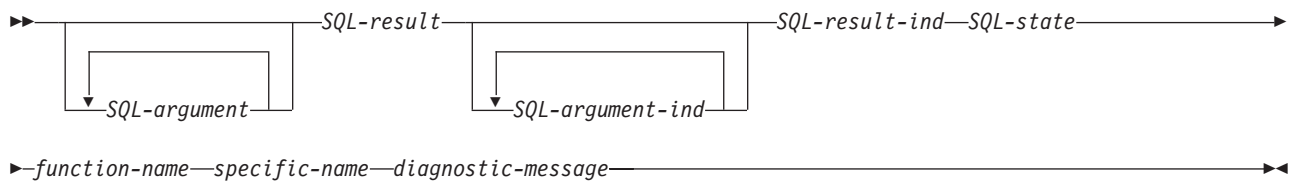
```
►►─┬─────────────────────────┬──────SQL-result──┬───────────────────────┬──────SQL-result-ind──SQL-state──────────────►
   │  ┌──────────────────┐   │                  │  ┌────────────────┐   │
   └──┴───SQL-argument───┴───┘                  └──┴──SQL-argument-ind──┴┘

►──function-name──specific-name──diagnostic-message─────────────────────────────────────────────────────────────────►◄
```

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.

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 called.
>
> 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 called. 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 called.
>
> 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 called. 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.

Related reference

SQL messages and codes

Parameter style DB2SQL:

With the DB2SQL parameter style, the same parameters and the same order of parameters are passed to an external program or a service program as with the SQL parameter style. However, DB2SQL allows additional optional parameters to be passed 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:

```
>>─┬───────────────────┬─SQL-result─┬───────────────────────┬─SQL-result-ind─SQL-state──────>
   │  ┌───────────────┐ │            │  ┌───────────────────┐ │
   └──▼──SQL-argument─┴─┘            └──▼──SQL-argument-ind─┴─┘

>──function-name──specific-name──diagnostic-message─┬───────────┬─┬───────────┬─┬────────┬──><
                                                    └─scratchpad┘ └─call-type─┘ └─dbinfo─┘
```

For table functions:

```
          ┌───────────────────────┐                        ┌────────────────┐
>>─┬───────────────┬─▼─SQL-result──┴─┬───────────────────┬─▼─SQL-result-ind──┴─SQL-state────>
   │  ┌──────────┐ │                 │  ┌──────────────┐ │
   └──▼─SQL-argument─┴─┘             └──▼─SQL-argument-ind─┴─┘
```

```
►─ function-name ── specific-name ── diagnostic-message ─┬──────────┬─ call-type ─┬─────────┬─────────►◄
                                                         └scratchpad┘             └ dbinfo ┘
```
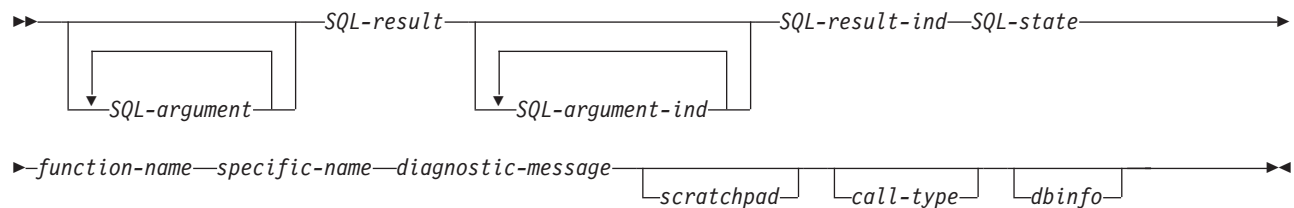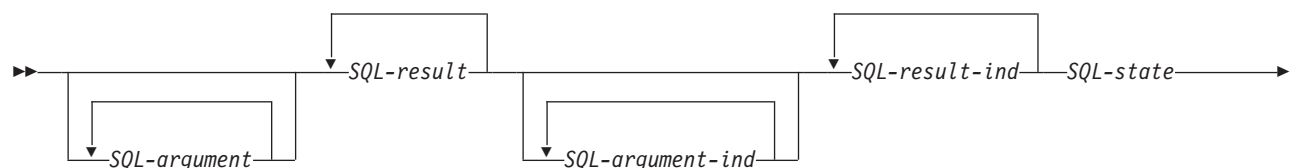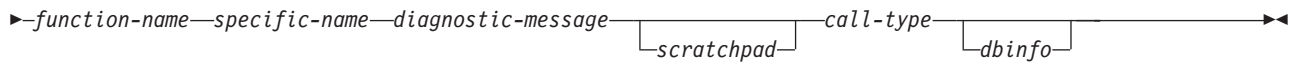
scratchpad

> 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 before 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 the system when returned 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 the system when returned 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:

With the GENERAL parameter style, the parameters are passed to an external service program just as they are specified in the CREATE FUNCTION statement. This parameter style can be used only with scalar user-defined functions (UDFs).

The format is:

```
►►──SQL-result = func──(──┬─────────────────┬──)────────────────────────►◄
                         │   ◄─────────┐    │
                         └─SQL-argument─┘
```

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 GENERAL WITH NULLS parameter style can be used only with scalar user-defined functions (UDFs).

With this parameter style, the parameters are passed into the service program as follows (in the order specified):
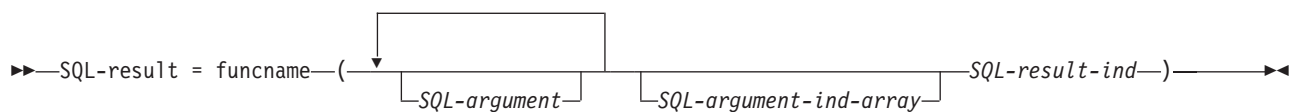
```
►►──SQL-result = funcname──(──┬─────────────────┬──┬──────────────────────┬──SQL-result-ind──)────►◄
                             │  ◄─────────┐    │  │                      │
                             └─SQL-argument─┘     └─SQL-argument-ind-array─┘
```

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 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 or without single quotation marks. 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 will not find the entry because it is in lowercase *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:

The DB2GENERAL parameter style is used by Java user-defined functions (UDFs).

Related concepts

Java SQL routines

Parameter style Java:

The Java parameter style is the style specified by the SQLJ Part 1: SQL Routines standard.

Related concepts

Java SQL routines

Table function considerations:

An external table function is a user-defined function (UDF) that delivers a table to the SQL statement in which it is referenced. A table function reference is valid only in a FROM clause of a SELECT statement.

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-ind* 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.

Error processing for UDFs:

When an error occurs in processing a user-defined function (UDF), the system follows a specified model to handle the error.

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.

Threads considerations:

A user-defined function (UDF) that is defined as FENCED runs in the same job as the SQL statement that calls the function. However, the UDF runs in a system thread, separate from the thread that is running the SQL statement.

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 called. 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.

   Related reference

   Multithreaded applications

   "Fenced or unfenced considerations" on page 167
   When you create a user-defined function (UDF), consider whether to make the UDF an unfenced UDF.

Parallel processing:

A user-defined function (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.

User-defined table functions cannot run in parallel; therefore, DISALLOW PARALLEL must be specified when creating the function

   Related reference

   Multithreaded applications

Fenced or unfenced considerations:

When you create 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 can 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;
```

> **Related reference**
>
> "Threads considerations" on page 166
> A user-defined function (UDF) that is defined as FENCED runs in the same job as the SQL statement that calls the function. However, the UDF runs in a system thread, separate from the thread that is running the SQL statement.

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 attributes of the function 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 function's attribute cannot 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 program object cannot be updated, the function will still be created.

Examples: UDF code

These examples show how to implement user-defined function (UDF) code by using SQL functions and external functions.

Example: Square of a number UDF:

Suppose that you want a function that returns the square of a number.

The query statement is:
```
SELECT SQUARE(myint) FROM mytable
```

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

The following examples show how to define the UDF in several different ways.

Using an SQL function

The CREATE FUNCTION statement:
```
CREATE FUNCTION SQUARE( inval INT) RETURNS INT
LANGUAGE SQL
SET OPTION DBGVIEW=*SOURCE
BEGIN
RETURN(inval*inval);
END
```

This creates an SQL function that you can debug.

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:

Suppose that you want to number the rows in a SELECT statement. So you write a user-defined function (UDF) that increments and returns a counter.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

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[92];
};

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:

Suppose that you write a table function that returns weather information for various cities in the United States.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

The weather date 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 */ (See note 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  */
/* need 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++ ;
   }
   /* can 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 */ (See note 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' ;
```

```c
/* 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 clean up (no real args!): Close table */ (See note 3)
    case SQL_TF_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 notes in this UDF code, note the following points:

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. 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)';
```

For this table function definition, note the following points:

- It does not take any input, and it returns seven 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.

- CARDINALITY 100, the default, is an estimate of the expected number of rows returned, which is 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.

To select all of the rows generated by this table function, use the following query:

```
SELECT *
    FROM TABLE (tfweather_u())x
```

Using UDFs in SQL statements

Scalar and column user-defined functions (UDFs) can be called within an SQL statement almost everywhere an expression is valid. Table UDFs can be called in the FROM clause of a SELECT statement. Listed here are a few restrictions on the use of UDFs.

- UDFs and system generated functions cannot be specified in check constraints. Check constraints also cannot contain references to some built-in functions that are implemented by the system as UDFs.
- 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 (ALWCPYDTA(*YES) or (*OPTIMIZE)).

Using parameter markers or the NULL values as function arguments:

You cannot use a parameter marker or a NULL value as a function argument. Because function resolution does not know what data type this type of argument might turn out to be, it cannot resolve the reference.

You cannot code the following:

```
BLOOP(?)
```

or

```
BLOOP(NULL)
```

You can use the CAST specification to provide a data type for the parameter marker or NULL value that function resolution can use:

```
BLOOP(CAST(? AS INTEGER))
```

or

```
BLOOP(CAST(NULL AS INTEGER))
```

Using qualified function references:

If you use a qualified function reference, you restrict the search for a matching function to the specified 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 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 chooses 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 and DB2 performs the appropriate data value conversions.

2. If for some reason you want to call the second BLOOP with a SMALLINT or INTEGER argument, you need to take an explicit action in your statement as follows:

   ```
   SELECT PABLO.BLOOP(DOUBLE(COLUMN1)) FROM T
   ```

3. If you want to call the first BLOOP with a DECIMAL or DOUBLE argument, you have your choice of explicit actions, depending on your intent:

   ```
   SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T
   SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
   ```

   Related reference

   "Using unqualified function references"
   You can use an unqualified function reference instead of a qualified function reference. When searching for a matching function, DB2 normally uses the function path to qualify the reference.

   "Defining a UDT" on page 200
   You define a user-defined type (UDT) using the CREATE DISTINCT TYPE statement.

Using unqualified function references:

You can use an unqualified function reference instead of a qualified function reference. When searching for a matching function, DB2 normally uses the function path to qualify the reference.

In the case of the DROP FUNCTION and COMMENT ON FUNCTION statements, 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 that 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 in the section Using qualified function reference, 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 there is a BLOOP function in schema KATHY, the function selection can very well resolve to that function, and your statement executes without error. You are not notified because DB2 assumes that you know what you are doing. It is your responsibility to identify the incorrect output from your statement and make the required correction.

   Related reference

   "Using qualified function references" on page 176
   If you use a qualified function reference, you restrict the search for a matching function to the specified schema.

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 functions, that have the given name, the same number of defined parameters as arguments, and 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.

An interesting feature is 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 aggregate functions are involved.

Refining an earlier example:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

Now consider the following 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'
  ```

  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 'ABCDEFG' 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 might not return the expected result since the returned integer will be cast as a VARCHAR before the CONCAT is performed. If the first BLOOP was not present, the other BLOOP is chosen and the statement execution is successful.

- UDFs can be defined with parameters or results having any of the LOB types: BLOB, CLOB, or DBCLOB. The system will materialize the entire LOB value in storage before calling 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). Referring to the FINDSTRING defined in "Example: String search" on page 157 both of the following are valid in the program:

```
... SELECT FINDSTRING (:clob150K, :string) FROM ...
... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
```

- External 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 before 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 external UDFs. Note that the argument to such a function can be any LOB value of the defined type; it does not need 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. 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 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 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 be careful with your function path. Otherwise your results may be unpredictable.

Triggers

A *trigger* is a set of actions that runs automatically when a specified change operation is performed on a specified table or view.

The change operation can be an SQL INSERT, UPDATE, or DELETE statement, or an insert, an update, or a 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 as SQL or external.

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 or view, the trigger support calls the trigger program whenever a change operation is initiated against the table or view, or any logical file or view created over the table or view. SQL triggers and external triggers can be defined for the same table. Only SQL triggers can be defined for a view. Up to 200 triggers can be defined for a single table or view.

Each change operation for a table 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)

Each change operation for a view can call an instead of trigger which will perform some set of actions instead of the insert, update, or delete. A view can be associated with an:
- Instead of delete trigger
- Instead of insert trigger
- Instead of update trigger

Related tasks

Triggering automatic events in your database

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 and views whenever an insert, an update, or a 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 on 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 or view that the trigger is defined on, is modified.

An SQL trigger can be created by specifying the CREATE TRIGGER SQL statement. All objects referred to in the CREATE TRIGGER statement (such as tables and functions) must exist; otherwise, the trigger will not be created. 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.

Related concepts

"Debugging an SQL routine" on page 186
By specifying SET OPTION DBGVIEW = *SOURCE in the CREATE PROCEDURE, CREATE FUNCTION, or CREATE TRIGGER statement, you can debug the generated program or module at the SQL statement level.

BEFORE SQL triggers:

BEFORE triggers cannot change tables, but they can be used to verify input column values and to change 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 before 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 is 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 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 is 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 before 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:

An after trigger runs after the corresponding insert, update, or delete changes are applied to the table.

The WHEN condition can be used in an SQL trigger to specify a condition. If the condition evaluates to true, the SQL statements in the SQL trigger routine body are run. If the condition evaluates to false, the SQL statements in the SQL trigger routine body are not run, 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 activated only 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.

INSTEAD OF SQL triggers:

An INSTEAD OF trigger is an SQL trigger that is processed "instead of" an SQL UPDATE, DELETE or INSERT statement. Unlike SQL BEFORE and AFTER triggers, an INSTEAD OF trigger can be defined only on a view, not a table.

An INSTEAD OF trigger allows a view, which is not inherently insertable, updatable, or deletable, to be inserted into, updated, or deleted from. See CREATE VIEW for more information about deleteable, updatable, and insertable views.

After an SQL INSTEAD OF trigger is added to a view, the view which previously could only be read from can be used as the target of an insert, update, or delete operation. The INSTEAD OF trigger defines the operations which need to be performed to maintain the view.

A view can be used to control access to tables. INSTEAD OF triggers can simplify the maintenance of access control to tables.

Using an INSTEAD OF trigger

The definition of the following view V1 is updatable, deletable, and insertable:

```
CREATE TABLE T1 (C1 VARCHAR(10), C2 INT)
CREATE VIEW V1(X1) AS SELECT C1 FROM T1 WHERE C2 > 10
```

For the following insert statement, C1 in table T1 will be assigned a value of 'A'. C2 will be assigned the NULL value. The NULL value would cause the new row to not match the selection criteria C2 > 10 for the view V1.

```
INSERT INTO V1 VALUES('A')
```

Adding the INSTEAD OF trigger IOT1 can provide a different value for the row that will be selected by the view:

```
CREATE TRIGGER IOT1 INSTEAD OF INSERT ON V1
REFERENCING NEW AS NEW_ROW
FOR EACH ROW MODE DB2SQL
INSERT INTO T1 VALUES(NEW_ROW.X1, 15)
```

Making a view deletable

The definition of the following join view V3 is not updatable, deletable, or insertable:

```
CREATE TABLE A (A1 VARCHAR(10), A2 INT)
CREATE VIEW V1(X1) AS SELECT A1 FROM A
```

```
CREATE TABLE B (B1 VARCHAR(10), B2 INT)
CREATE VIEW V2(Y1) AS SELECT  B1 FROM B

CREATE VIEW V3(Z1, Z2) AS SELECT V1.X1, V2.Y1 FROM V1, V2 WHERE V1.X1 = 'A' AND V2.Y1 > 'B'
```

Adding the INSTEAD OF trigger IOT2 makes the view V3 deletable:

```
CREATE TRIGGER IOT2 INSTEAD OF DELETE ON V3
REFERENCING OLD AS OLD_ROW
FOR EACH ROW MODE DB2SQL
BEGIN
  DELETE FROM A WHERE A1 = OLD_ROW.Z1;
  DELETE FROM B WHERE B1 = OLD_ROW.Z2;
END
```

With this trigger, the following DELETE statement is allowed. It deletes all rows from table A having an A1 value of 'A', and all rows from table B having a B1 value of 'X'.

```
DELETE FROM V3 WHERE Z1 = 'A' AND Z2 = 'X'
```

INSTEAD OF triggers with views defined on views

The following definition of view V2 defined on V1 is not inherently insertable, updatable, or deletable:

```
CREATE TABLE T1 (C1 VARCHAR(10), C2 INT)
CREATE TABLE T2 (D1 VARCHAR(10), D2 INT)
CREATE VIEW V1(X1, X2) AS SELECT C1, C2 FROM T1
UNION SELECT D1, D2 FROM T2

CREATE VIEW V2(Y1, Y2) AS SELECT X1, X2 FROM V1
```

Adding the INSTEAD OF trigger IOT1 to V1 does not make V2 updatable:

```
CREATE TRIGGER IOT1 INSTEAD OF UPDATE ON V1
REFERENCING OLD AS OLD_ROW NEW AS NEW_ROW
FOR EACH ROW MODE DB2SQL
BEGIN
  UPDATE T1 SET C1 = NEW_ROW.X1, C2 = NEW_ROW.X2 WHERE
    C1 = OLD_ROW.X1 AND C2 = OLD_ROW.X2;
  UPDATE T2 SET D1 = NEW_ROW.X1, D2 = NEW_ROW.D2 WHERE
    D1 = OLD_ROW.X1 AND D2 = OLD_ROW.X2;
END
```

View V2 remains not updatable since the original definition of view V2 remains not updatable.

Using INSTEAD OF triggers with BEFORE and AFTER triggers

The addition of an INSTEAD OF trigger to a view does not cause any conflicts with BEFORE and AFTER triggers defined on the base tables:

```
CREATE TABLE T1 (C1 VARCHAR(10), C2 DATE)
CREATE TABLE T2 (D1 VARCHAR(10))

CREATE TRIGGER AFTER1 AFTER DELETE ON T1
REFERENCING OLD AS OLD_ROW
FOR EACH ROW MODE DB2SQL
  DELETE FROM T2 WHERE D1 = OLD_ROW.C1

CREATE VIEW V1(X1, X2) AS SELECT SUBSTR(T1.C1, 1, 1), DAYOFWEEK_ISO(T1.C2) FROM T1

CREATE TRIGGER IOT1 INSTEAD OF DELETE ON V1
REFERENCING OLD AS OLD_ROW
FOR EACH ROW MODE DB2SQL
  DELETE FROM T1 WHERE C1 LIKE (OLD_ROW.X1 CONCAT '%')
```

Any delete operations for view V1 result in the AFTER DELETE trigger AFTER1 being activated also because trigger IOT1 performs a delete on table T1. The delete for table T1 causes the AFTER1 trigger to be activated.

Dependent views and INSTEAD OF triggers

When adding an INSTEAD OF trigger to a view, if the view definition references views that also have INSTEAD OF triggers defined, you should define INSTEAD OF triggers for all three operations, UPDATE, DELETE, and INSERT, to avoid confusion on what capabilities the view being defined contains versus what the capabilities of any dependent views have.

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 processing 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 runs, 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 might need to refer to all of the affected rows for an SQL insert, update, or delete operation. For example, a 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 processed a single time following the processing 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 that contains 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 Add Physical File Trigger (ADDPFTRG) CL command or you can add it using System i Navigator. To add a trigger to a table, you must do the following things:

* Identify the table.
* Identify the kind of operation.
* Identify the program that performs the actions that you want.

> **Related tasks**
>
> Triggering automatic events in your database

Debugging an SQL routine

By specifying SET OPTION DBGVIEW = *SOURCE in the CREATE PROCEDURE, CREATE 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 source file QSQDSRC in the routine library. If the library cannot be determined, QSQDSRC is created in QTEMP. 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.

Improving performance of procedures and functions

When creating a stored procedure or a user-defined function (UDF), the SQL procedural language processor does not always generate the most efficient code. However, you can reduce the number of database engine calls and improve performance.

Some changes are in the design of a routine and some are in the implementation. For example, differences between how the C language compiler handles host variables and the way the SQL procedural processor requires the host variables to be handled can cause many calls to the database engine. These calls are very expensive and, when done many times, can significantly degrade performance.

Improving implementation of procedures and functions

These coding techniques help reduce the processing time of a function or procedure.

The following tips are especially important for functions because a function tends to be called multiple times from many different procedures:
- Use the NOT FENCED option so UDFs run in the same thread as the caller.
- Use the DETERMINISTIC option on procedures and UDFs that return the same results for identical inputs. This allows the optimizer to cache the results of a function call or order where the function is called in the execution stream to reduce the run time.
- Use the NO EXTERNAL ACTION option on UDFs that do not take an action outside the scope of the function. An example of an external action is a function that initiates a different process to fulfill a transaction request.

Coding techniques used for the SQL routine body can have a major impact on the runtime performance of the generated C program. By writing your routine to allow greater use of C code for assignments and comparisons, the overhead of an equivalent SQL statement is avoided. The following tips should help your routine generate more C code and fewer SQL statements.

- Declare host variables as NOT NULL when possible. This saves the generated code from having to check and set the null value flags. Do not automatically set all variables to NOT NULL. When you specify NOT NULL, you need to also give a default value. If a variable is always used in the routine, a default value might help. However, if a variable is not always used, having a default value set may cause additional initialization overhead that is not needed. A default value is best for numeric values, where an additional database call to process the assignment of the default value is not needed.
- Avoid character and date data types when possible. An example of this is a variable used as a flag with a value of 0, 1, 2, or 3. If this value is declared as a single character variable instead of an integer, it causes calls to the database engine that can be avoided.
- Use integer instead of decimal with zero scale, especially when the variable is used as a counter.
- Do not use temporary variables. Look at the following example:

```
IF M_days<=30 THEN
 SET I = M_days-7;
 SET J = 23
 RETURN decimal(M_week_1 + ((M_month_1 - M_week_1)*I)/J,16,7);
END IF
```

This example can be rewritten without the temporary variables:

```
IF M_days<=30 THEN
 Return decimal(M-week_1 + ((M_month_1 - M_week_1)* (M_days-7))/23,16,7);
END IF
```

- Combine sequences of complex SET statements into one statement. This applies to statements where C code only cannot be generated because of CCSIDS or data types.

```
SET var1 = function1(var2);
SET var2 = function2();
```

Can be rewritten into one statement:

```
SET var1 = function1(var2), var2 = function2();
```

- Use IF () ELSE IF () ... ELSE ... constructs instead of IF (x AND y) to avoid unnecessary comparisons.
- Do as much in SELECT statements as possible:

```
SELECT A INTO Y FROM B;
 SET Y=Y||'X';
```

Rewrite this example:

```
SELECT A || 'X' INTO Y FROM B
```

- Avoid doing character or date comparisons inside of loops when not necessary. In some cases the loop can be rewritten to move a comparison to precede the loop and have the comparison set an integer variable that is used within the loop. This causes the complex expression to be evaluated only one time. An integer comparison within the loop is more efficient since it can be done with generated C code.
- Avoid setting variables that might not be used. For example, if a variable is set outside of the an IF statement, be sure that the variable will actually be used in all instances of the IF statement. If not, then set the variable only in the portion of the IF statement that is it actually used.
- Replace sections of code with a single SELECT statement when possible. Look at the following code snippet:

```
SET vnb_decimal = 4;
cdecimal:
   FOR vdec AS cdec CURSOR FOR
   SELECT nb_decimal
   FROM K$FX_RULES
   WHERE first_currency=Pi_curl AND second_currency=P1_cur2
   DO
    SET vnb_decimal=SMALLINT(cdecimal.nb_decimal);
END FOR cdecimal;

 IF vnb_decimal IS NULL THEN
```

```
   SET vnb_decimal=4;
 END IF;
 SET vrate=ROUND(vrate1/vrate2,vnb_decimal);
 RETURN vrate;
```

This code snippet can be more efficient if rewritten in the following way:

```
RETURN( SELECT
    CASE
     WHEN MIN(nb_decimal) IS NULL THEN ROUND(Vrate1/Vrate2,4)
    ELSE ROUND(Vrate1/Vrate2,SMALLINT(MIN(nb_decimal)))
    END
    FROM K$FX_RULES
    WHERE first_currency=Pi_curl AND second_currency=Pi_cur2);
```

* C code can only be used for assignments and comparisons of character data if the CCSIDs of both operands are the same, if one of the CCSIDs is 65535, if the CCSID is not UTF8, and if truncation of character data is not possible. If the CCSID of the variable is not specified, the CCSID is not determined until the procedure is called. In this case, code must be generated to determine and compare the CCSID at runtime. If an alternate collating sequence is specified or if *JOBRUN is specified, C code cannot be generated for character comparisons.

* Use the same data type, length and scale for numeric variables that are used together in assignments. C code can only be generated if truncation is not possible.

```
DECLARE v1, v2 INT;
SET v1 = 100;
SET v1 =  v2;
```

Redesigning routines for performance

Even if you follow all of the implementation tips, a procedure or function still might not perform well. In this case, you need to look at the design of the procedure or user-defined function (UDF) and see whether there are any changes that can be made to improve the performance.

Here are the different types of design changes that you can look at.

The first change is to reduce the number of database calls or function calls that a procedure makes, a process similar to looking for blocks of code that can be converted to SQL statements. Many times you can reduce the number of calls by adding additional logic to your code.

A more difficult design change is to restructure a whole function to get the same result a different way. For example, your function uses a SELECT statement to find a route that meets a particular set of criteria and then executes that statement dynamically. By looking at the work that the function is performing, you might be able to change the logic so that the function can use a static SELECT query to find the answer, thereby improving your performance.

You should also use nested compound statements to localize exception handling and cursors. If several specific handlers are specified, code is generated to check to see if the error occurred after each statement. Code is also generated to close cursors and process savepoints if an error occurs in a compound statement. In routines with a single compound statement with multiple handlers and multiple cursors, code is generated to process each handler and cursor after every SQL statement. If you scope the handlers and cursors to a nested compound statement, the handlers and cursors are only checked within the nested compound statement.

In the following routine, code to check the SQLSTATE '22H11' error will only be generated for the statements within the lab2 compound statement. Specific checking for this error will not be done for any statements in the routine outside of the lab2 block. Code to check the SQLEXCEPTION error will be generated for all statements in both the lab1 and lab2 blocks. Likewise, error handling for closing cursor c1 will be limited to the statements in the lab2 block.

```
Lab1: BEGIN
   DECLARE var1 INT;
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
        RETURN -3;
   lab2:  BEGIN
      DECLARE EXIT HANDLER FOR SQLSTATE '22H11'
           RETURN -1;
      DECLARE c1 CURSOR FOR SELECT col1 FROM table1;
      OPEN c1;
      CLOSE c1;
   END lab2;
END Lab1
```

Because redesigning a whole routine takes a lot of effort, examine routines that are showing up as key performance bottlenecks rather than looking at the application as a whole. More important than redesigning existing performance bottlenecks is to spend time during the design of the application thinking about the performance impacts of the design. Focusing on areas of the application that are expected to be high use areas and making sure that they are designed with performance in mind saves you from having to do a redesign of those areas later.

Processing special data types

Most data types, such as INTEGER and CHARACTER, do not have any special processing characteristics. However, a few data types require special functions or locators to use them effectively.

Large objects

A large object (LOB) is a string data type with a size ranging from 0 bytes to 2 GB (GB equals 1 073 741 824 bytes).

The VARCHAR, VARGRAPHIC, and VARBINARY data types have a limit of 32 KB (where KB equals 1024 bytes) of storage. While this might be sufficient for small to medium-sized text data, applications often need to store large text documents. They might also need to store a wide variety of additional data types, such as audio, video, drawings, mixed text and graphics, and images. Some data types can store these data objects as strings of up to 2 GB.

These data types are binary large objects (BLOBs), single-byte character large objects (CLOBs), and double-byte character large objects (DBCLOBs). Each table can have a large amount of associated LOB data. Although a single row that contains one or more LOB values cannot exceed 3.5 GB, a table can contain nearly 256 GB of LOB data.

You can refer to and manipulate LOBs using host variables as you do any other data type. However, host variables use the program's storage that might not be large enough to hold LOB values, so you might need to manipulate large values in other ways. *Locators* are useful for identifying and manipulating 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.

Large object data types
Here are the definitions of binary large objects (BLOBs), character large objects (CLOBs), and double-byte character large objects (DBCLOBs).

- Binary large objects (BLOBs) — A binary string made up of bytes with no associated code page. This data type can store binary data larger than VARBINARY (32K limit). This data type is good for storing image, voice, graphical, and other types of business or application-specific data.

- Character large objects (CLOBs) — A character string made up of single-byte characters with an associated code page. This data type is appropriate for storing text-oriented information where the amount of information can grow beyond the limits of a regular VARCHAR data type (upper limit of 32K bytes). Code page conversion of the information is supported.

- Double-byte character large objects (DBCLOBs) — A character string made up of double-byte characters with an associated code page. This data type is appropriate for storing text-oriented information where double-byte character sets are used. Again, code page conversion of the information is supported.

Large object locators

A large object (LOB) locator is a small, easily managed value that is used 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 held in the database server. Using a LOB locator, a program can manipulate the LOB value as if it were 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, 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 table(s) that 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 rollback operation frees all LOB locators associated with the transaction.

LOB locators can also be passed to and returned from UDFs. Within the UDF, those functions that work on LOB data can be used 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 file.

How a LOB value is used 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.

> **Related reference**
> "LOB file reference variables" on page 195
> File reference variables are similar to host variables except that they are used to transfer data to and from integrated file system files (not to and from memory buffers).
> "Example: Using a locator to work with a CLOB value"
> Suppose that you want your application program to retrieve a locator for a character large object (CLOB) value and then use the locator to extract data from the CLOB value.

Example: Using a locator to work with a CLOB value

Suppose that you want your application program to retrieve a locator for a character large object (CLOB) value and then use the locator to extract data from the CLOB value.

Using this method, the program allocates only enough storage for one piece of CLOB data (the size is determined by the program). In addition, the program needs to issue only one fetch call using the cursor.

How the example LOBLOC program works

The example program has three sections that work with the CLOB locator variables:

1. **Declare host variables.** CLOB locator host variables are declared.
2. **Fetch the CLOB value into the locator host variable.** A CURSOR and FETCH routine is used to obtain the location of a CLOB field in the database to a locator host variable.
3. **Free the CLOB locators.** The CLOB locators used in this example are freed, releasing the locators from their previously associated values.

Related concepts

"Large object locators" on page 191
A large object (LOB) locator is a small, easily managed value that is used to refer to a much larger value.

Example: LOBLOC in C:

This example program, written in C, uses a locator to retrieve a CLOB value.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
      char number[7];
      long deptInfoBeginLoc;
      long deptInfoEndLoc;
      SQL TYPE IS CLOB_LOCATOR resume;       [1]
      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;
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
   }
   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;
|
|      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;
|
|            /* Locate the beginning of "Education" section (end of "Dept.Info" */
|            EXEC SQL VALUES (POSSTR(:resume, 'Education'))
|                INTO :deptInfoEndLoc;
|
|            /* Obtain ONLY the "Department Information" section by using SUBSTR */
|            EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
|                :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
|
|            /* Append the "Department Information" section to the :buffer var. */
|            EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
|         } /* endif */
|      } while ( 1 );
|
|      printf ("%s\n",buffer);
|
|      EXEC SQL FREE LOCATOR :resume, :deptBuffer; [3]
|
|      EXEC SQL CLOSE c1;
|
|      EXEC SQL CONNECT RESET;
|      return 0;
| }
| /* end of program : LOBLOC */
```

Example: LOBLOC in COBOL:

This example program, written in COBOL, uses a locator to retrieve a CLOB value.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

```
|  Identification Division.
|  Program-ID. "lobloc".
|
|  Data Division.
|  Working-Storage Section.
|      EXEC SQL INCLUDE SQLCA END-EXEC.
|
|      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
|  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.    [1]
|  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.
|
|  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.
|
| * 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 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]
|
|       EXEC SQL CLOSE c1 END-EXEC.
|
|       EXEC SQL CONNECT RESET END-EXEC.
|   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.
|
| * Locate the beginning of "Education" section (end of Dept.Info)
|       EXEC SQL VALUES (POSSTR(:resume, 'Education'))
|               INTO :di-end-loc END-EXEC.
```

```
|        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.
|
| * Append the "Department Information" section to the :buffer var
|        EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
|                   END-EXEC.
|
|        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 integrated file system 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 helps move LOB data. Programs use file reference variables to transfer LOB data from the integrated file system file directly to the database engine. To carry out the movement of LOB data, the application does not need 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 is 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 requester 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.

Related concepts

"Large object locators" on page 191
A large object (LOB) locator is a small, easily managed value that is used to refer to a much larger value.

Integrated file system

Example: Extracting CLOB data to a file

Suppose that you need to retrieve character large object (CLOB) elements from a table into an external file.

How the example LOBFILE program works

The example program has three sections that work with the CLOB file reference variable:
1. **Declare host variables.** A CLOB file reference host variable is declared. The declaration is expanded as a structure that includes declarations for a file name, file name length, and file options.
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 path name 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 file name that is referenced by the host variable.

Example: LOBFILE in C:

This example program, written in C, extracts CLOB data from a table to an external file.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION;
      SQL TYPE IS CLOB_FILE resume;        [1]
      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;
|
|     else if (argc == 3) {
|        strcpy (userid, argv[1]);
|        strcpy (passwd, argv[2]);
|        EXEC SQL CONNECT TO sample USER :userid USING :passwd;
|
|     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;
|     return 0;
| }
| /* end of program : LOBFILE */
```

Example: LOBFILE in COBOL:

This example program, written in COBOL, extracts CLOB data from a table to an external file.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

```
|
|  Identification Division.
|  Program-ID. "lobfile".
|
|  Data Division.
|  Working-Storage Section.
|      EXEC SQL INCLUDE SQLCA END-EXEC.
|
|      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
|  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.  [1]
|  01 lobind            pic s9(4) comp-5.
|      EXEC SQL END DECLARE SECTION END-EXEC.
|
|  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 "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.
|  End-Prog.
|             stop run.
```

Example: Inserting data into a CLOB column

This example shows how to insert data from a regular file referenced by :hv_text_file into a character
large object (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 want to insert into the table.
* clobtab is the name of the table with the CLOB data type.

```
    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);
```

Displaying the layout of LOB columns

When you use CL commands, such as Display Physical File Member (DSPPFM), to display a row of data
from a table that holds large object (LOB) columns, the LOB data stored in the row is not displayed.
Instead, the database shows a special value for the LOB columns.

The layout of this special value is as follows:
* 13 to 28 bytes of hexadecimal 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 looks as follows.

- For ColumnOne: 10 bytes filled with character data.
- For ColumnTwo: 22 bytes filled with hexadecimal zeros and 16 bytes filled with '*POINTER '.
- For ColumnThree: 16 bytes filled with hexadecimal 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

These commands return a buffer that gives the user addressability to the large object (LOB) data that has 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.

User-defined distinct types

A user-defined distinct type (UDT) is a mechanism to extend DB2 capabilities beyond the built-in data types that are available.

User-defined distinct types enable you to define new data types to DB2, which gives you considerable power because 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.

These benefits are associated with UDTs:
- **Extensibility**.

By defining new types, you can indefinitely increase the set of types provided by DB2 to support your applications.

- **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.

- **Consistent behavior**.

  Strong typing ensures that your UDTs will behave appropriately. It guarantees that only functions defined on your UDT can be applied to instances of the UDT.

- **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.

- **Extensible behavior**.

  The definition of user-defined functions on types can augment the functionality provided to manipulate your UDT at any time.

- **Foundation for object-oriented extensions**.

  UDTs are the foundation for most object-oriented features. They represent the most important step toward object-oriented extensions.

    Related concepts

    "User-defined types" on page 12
    A *user-defined type* is a distinct data type that you can define independently of the data types that are provided by the database management system.

Defining a UDT

You define a user-defined type (UDT) using the CREATE DISTINCT TYPE statement.

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 the system 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, the system 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.

These functions are important for the manipulation of UDTs in queries.

The function path is used to resolve any references to an unqualified type name or function, except if the type name or function is the main object of a CREATE, DROP, or COMMENT ON statement.

    Related reference

    "Using qualified function references" on page 176
    If you use a qualified function reference, you restrict the search for a matching function to the specified schema.
    CREATE TYPE

Example: Money:

Suppose that you are writing applications that handle different currencies and want 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 EURO AS DECIMAL (9,2)
```

Example: Resumé:

Suppose that you want to keep the application forms that are filled out by applicants to your company in a table, and that you are going to use functions to extract 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 user-defined types (UDTs), you can start defining tables with columns whose types are UDTs.

Example: Sales:

Suppose that you want to define tables to keep your company's sales in different countries.

You create the tables 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         EURO)
```

The UDTs in the preceding examples are created with the same CREATE DISTINCT TYPE statements in "Example: Money" on page 200. Note that the preceding examples use check constraints.

Example: Application forms:

Suppose that you need to define a table to keep the forms that are filled out by applicants.

Create the table 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

Strong typing is an important concept associated with user-defined types (UDTs). Strong typing guarantees that only functions and operators defined on a 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 Euros 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 need to cast the instances of one or the other type. In the same sense, you need 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.

Examples: Using UDTs

These examples show the use of user-defined types (UDTs) in various situations.

Example: Comparisons between UDTs and constants:

Suppose that you want to know which products had total sales of over US $100 000.00 in the U.S. in the month of July 1998.

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > US_DOLLAR (100000)
  AND    month = 7
  AND    year  = 1998
```

Because you cannot compare U.S. dollars with instances of the source type of U.S. dollars (that is, DECIMAL) directly, you have used the cast function provided by DB2 to cast from DECIMAL to U.S. dollars. You can also use the other cast function provided by DB2 (that is, the one to cast from U.S. 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. You might have written the above query as:

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > CAST (100000 AS us_dollar)
  AND    MONTH = 7
  AND    YEAR  = 1998
```

Example: Casting between UDTs:

Suppose that you want to define a user-defined function (UDF) that converts Canadian dollars to U.S. dollars.

You can obtain the current exchange rate from a file managed outside of DB2. 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. 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 Euros to U.S. dollars is similar to the example above:

```
CREATE FUNCTION EURO_TO_US_DOUBLE(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/CURRENCIES(C_EURO_US)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC

CREATE FUNCTION EURO_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  SOURCE EURO_TO_US_DOUBLE(DOUBLE)

CREATE FUNCTION US_DOLLAR(EURO) RETURNS US_DOLLAR
  SOURCE EURO_TO_US_DEC (DECIMAL())
```

Example: Comparisons involving UDTs:

Suppose that you want to know which products had higher sales in the U.S. than in Canada and Germany for the month of March 2003.

Issue the following SELECT statement:

```
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   = 2003
  AND CDN.MONTH = 3
  AND CDN.YEAR   = 2003
  AND GERMAN.MONTH = 3
  AND GERMAN.YEAR  = 2003
```

Because you cannot directly compare U.S. dollars with Canadian dollars or Euros, you use the UDF to cast the amount in Canadian dollars to U.S. dollars, and the UDF to cast the amount in Euros to U.S. 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 that you have defined a sourced user-defined function (UDF) on the built-in SUM function to support SUM on Euros.

The function statement is as follows:
```
CREATE FUNCTION SUM (EURO)
  RETURNS EURO
  SOURCE SYSIBM.SUM (DECIMAL())
```

You want to know the total of sales in Germany for each product in the year of 2004. You want to obtain the total sales in U.S. dollars:
```
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 2004
  GROUP BY PRODUCT_ITEM
```

You cannot write **SUM** (US_DOLLAR (TOTAL)), unless you had defined a SUM function on U.S. dollar in a manner similar to the above.

> **Related reference**
>
> "Example: Assignments involving different UDTs" on page 205
> Suppose that you have defined these sourced user-defined functions (UDFs) on the built-in SUM function to support SUM on U.S. and Canadian dollars.

Example: Assignments involving UDTs:

Suppose that you want to store the form that is 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.

Related reference

"Example: Assignments in dynamic SQL"
If you want to store the application form using dynamic SQL, you can use parameter markers.

Example: Assignments in dynamic SQL:

If you want to store the application form using dynamic SQL, you can use parameter markers.

The statement is 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.

Related reference

"Example: Assignments involving UDTs" on page 204
Suppose that you want to store the form that is filled out by a new applicant into the database.

Example: Assignments involving different UDTs:

Suppose that you have defined these sourced user-defined functions (UDFs) on the built-in SUM function to support SUM on U.S. and Canadian dollars.

```
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 U.S. dollars of each product and in each country, in separate tables:

```
CREATE TABLE US_SALES_04
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE GERMAN_SALES_04
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES_04
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)
```

```
    INSERT INTO US_SALES_04
      SELECT PRODUCT_ITEM, SUM (TOTAL)
      FROM US_SALES
      WHERE YEAR = 2004
      GROUP BY PRODUCT_ITEM

    INSERT INTO GERMAN_SALES_04
      SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
      FROM GERMAN_SALES
      WHERE YEAR = 2004
      GROUP BY PRODUCT_ITEM

    INSERT INTO CANADIAN_SALES_04
      SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
      FROM CANADIAN_SALES
      WHERE YEAR = 2004
      GROUP BY PRODUCT_ITEM
```

You explicitly cast the amounts in Canadian dollars and Euros to U.S. 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.

> **Related reference**
>
> "Example: Sourced UDFs involving UDTs" on page 204
> Suppose that you have defined a sourced user-defined function (UDF) on the built-in SUM function to support SUM on Euros.

Example: Using UDTs in UNION:

Suppose that you want to provide your U.S. users with a query to show the sales of every product of your company.

The SELECT statement is as follows:

```
    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 U.S. dollars and Euros to U.S. 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.

Examples: Using UDTs, UDFs, and LOBs

These examples show how to use user-defined types (UDTs), user-defined functions (UDFs), and large objects (LOBs) together in complex applications.

Example: Defining the UDT and UDFs

Suppose that you want to keep the electronic mail (e-mail) that is sent to your company in a table.

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 the LOB function to populate the database

Suppose that you populate your table by transferring your e-mail that is maintained in files into the DB2 for i5/OS database.

Run the following INSERT statement multiple times with different values of the HV_EMAIL_FILE until you have stored all your e-mail:

```
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 allows you to 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 that you need to know how much e-mail was sent by a specific customer regarding customer orders and you have the e-mail addresses of your customers in the customers table.

The statement is as follows:

```
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 that is capable of analyzing the contents of the message searching for relevant keywords or synonyms.

Example: Using LOB locators to manipulate UDT instances

Suppose that you want to obtain information about specific e-mail without having to transfer the entire e-mail into a host variable in your application program.

Remember that e-mail can be quite large. Because your user-defined type (UDT) is defined as a large object (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, and so on. 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 SQL scalar functions to get back the path to the object and the server on which the object is stored (see Built-in functions in the SQL Reference). 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.

A 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 a 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 only want the SQL table to contain references to the objects, not the actual bytes of storage. A good solution is to use DataLinks. The SQL table uses traditional SQL data types to contain information about each clip, such as title, length, date, and so on. But the clip itself is referenced using a DataLink column. Each row in the table stores 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 of using this technique:
* The integrated file system can store any type of stream file.
* The integrated file system can store extremely large objects, that does 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, and so on.

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 are 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.

Linking control levels in DataLinks

You can create a DataLink column with different link controls.

NO LINK CONTROL:

If a DataLink column is created with NO LINK CONTROL, no linking 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 FS permissions:

If a DataLink column is created with FILE LINK CONTROL with file system (FS) permissions, the system verifies that each DataLink value is a valid URL, with a valid server name and file name.

The file must exist when the row is being inserted into the SQL table. When the object is found, it is marked as linked. This means that the object cannot be moved, deleted, or renamed during the time 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 that contains 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 DB permissions:

If a DataLink column is created with FILE LINK CONTROL with database (DB) 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, and so on.) 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.

Working with DataLinks

To work with DataLinks, you need to understand the DataLink processing environment.

Support for the DataLink data type can be broken down into the following 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 metadata for each file. This code handles linking, unlinking, and commitment control issues. An important aspect of DataLinks is that the DLFM does not need to 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 run 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. Because there is extra overhead in this filter process, it is only run when the accessed object exists in one of the directories within a DataLink prefix. See the following discussion 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 includes 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 ENDTCPSVR *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 open 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 Data Links File Manager (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 Add Prefix to DataLink File Manager (ADDPFXDLFM) command.

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. The following links for URLs are valid because their paths have valid prefixes:

```
http://TESTSYS1/mydir/datalinks/videos/file1.mpg
```

or

```
file://TESTSYS1/mydir/datalinks/text/story1.txt
```

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 run if there are no linked objects anywhere in the directory structure contained within the prefix name.

Notes:

1. The following directories, or any of their subdirectories, should not be used as prefixes for DataLinks:
   - /QIBM
   - /QReclaim
   - /QSR
   - /QFPNWSSTG
2. Additionally, common base directories such as the following should not be used unless the prefix is a subdirectory within one of the base directories:
   - /home
   - /dev
   - /bin
   - /etc
   - /tmp
   - /usr
   - /lib

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 Add Host Database to DataLink File Manager (ADDHDBDLFM) command. 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 TESTDB or PRODDB library to be allowed to link objects on this server. Use the following:

```
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.

Using SQL in different environments

You can use SQL in many different environments.

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.

    Related reference

    "Updating data as it is retrieved from a table" on page 89
    You can update rows of data as you retrieve them by using a cursor.

    CLOSE

    DECLARE CURSOR

    DELETE

    FETCH

    UPDATE

Types of cursors

SQL supports serial and scrollable cursors. The type of cursor determines the positioning methods that can be used with the cursor.

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 n in relationship to the cursor's current position. For example, if n is -1, the cursor is positioned on the previous row of the result table. If n 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.

Examples: Using a cursor

These examples show the SQL statements that you can include in a program to define and work with a serial and a scrollable cursor.

Suppose that your program examines data about people in department D11. You can use either a serial or a scrollable cursor 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 41. A serial cursor example

Serial cursor SQL statement	Described in section
	"Step 1: Defining the cursor" on page 216.

EXEC SQL
 DECLARE THISEMP CURSOR FOR
  SELECT EMPNO, LASTNAME,
   WORKDEPT, JOB
   FROM CORPDATA.EMPLOYEE
   FOR UPDATE OF JOB
END-EXEC.	

	"Step 2: Opening the cursor" on page 217.
EXEC SQL OPEN THISEMP END-EXEC.	

Table 41. A serial cursor example (continued)

Serial cursor SQL statement	Described in section

EXEC SQL
  WHENEVER NOT FOUND
    GO TO CLOSE-THISEMP
END-EXEC.
	"Step 3: Specifying what to do when the end of data is reached" on page 217.

EXEC SQL
  FETCH THISEMP
    INTO :EMP-NUM, :NAME2,
      :DEPT, :JOB-CODE
END-EXEC.
	"Step 4: Retrieving a row using a cursor" on page 218.

... for all employees
    in department D11, update
    the JOB value:
 	
EXEC SQL
  UPDATE CORPDATA.EMPLOYEE
    SET JOB = :NEW-CODE
    WHERE CURRENT OF THISEMP
END-EXEC.
 	
... then print the row.
	"Step 5a: Updating the current row" on page 218.

... for other employees,
    delete the row:
 	
EXEC SQL
  DELETE FROM CORPDATA.EMPLOYEE
    WHERE CURRENT OF THISEMP
END-EXEC.
	"Step 5b: Deleting the current row" on page 219.

Branch back to fetch and process the next row.	
CLOSE-THISEMP. EXEC SQL **CLOSE** THISEMP END-EXEC.	"Step 6: Closing the cursor" on page 219.

For the scrollable cursor example, the program uses the RELATIVE position option to obtain a representative sample of salaries from department D11.

Table 42. A scrollable cursor example

Scrollable cursor SQL statement	Described in section

EXEC SQL
  DECLARE THISEMP **DYNAMIC SCROLL CURSOR FOR**
    SELECT EMPNO, LASTNAME,
      SALARY
      FROM CORPDATA.EMPLOYEE
      WHERE WORKDEPT = 'D11'
END-EXEC.
	"Step 1: Defining the cursor" on page 216.

Table 42. A scrollable cursor example (continued)

Scrollable cursor SQL statement	Described in section
EXEC SQL **OPEN** THISEMP END-EXEC.	"Step 2: Opening the cursor" on page 217.

EXEC SQL
  WHENEVER NOT FOUND
    GO TO CLOSE-THISEMP
END-EXEC.
	"Step 3: Specifying what to do when the end of data is reached" on page 217.

...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.
	"Step 4: Retrieving a row using a cursor" on page 218.

...calculate the average salary	
CLOSE-THISEMP. EXEC SQL **CLOSE** THISEMP END-EXEC.	"Step 6: Closing the cursor" on page 219.

Step 1: Defining the cursor:

To define a cursor to access the result table, 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  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, the ORDER BY clause, and the result table is not read-only and the cursor is not scrollable, 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.

For information about when the result table and cursor are read-only, see DECLARE CURSOR in the SQL reference topic collection.

Step 2: Opening 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: Specifying what to do when the end of data is reached:

The end-of-data condition occurs when the FETCH statement has retrieved the last row in the result table and your program issues a subsequent FETCH statement.

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).

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: Retrieving a row using a cursor:

To move the contents of a selected row into the host variables of your program, 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: Updating the current row:

When your program has positioned the cursor on a row, you can update the row by using the UPDATE statement with the WHERE CURRENT OF clause. The WHERE CURRENT OF clause specifies a cursor that points to the row that 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: Deleting the current row:

When your program has retrieved the current row, you can delete the row by using the DELETE statement with the WHERE CURRENT OF clause. The WHERE CURRENT OF clause specifies a cursor that points to the row that 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.

You can 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, and then delete it.

Step 6: Closing the cursor:

If you have processed the rows of a result table using a serial cursor, and you want to use the cursor again, issue a CLOSE statement to close the cursor before opening it again.

The statement looks like this:
```
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 statement. The program controls the blocking of rows by the number of rows requested on the FETCH statement (The Override Database File (OVRDBF) command has no effect.).

The maximum number of rows that can be requested on a single fetch call is 32 767. After 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. 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.

   Related concepts

   Embedded SQL programming

Multiple-row FETCH using a host structure array:

To use the multiple-row FETCH statement 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:

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

```
     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

    Related reference

    SQLCA (SQL communication area)

Multiple-row FETCH using a row storage area:

Before using a multiple-row FETCH statement with the row storage area, the application must define a row storage area and an associated description area.

The row storage area is a host variable defined in the application. The row storage area contains the results of the multiple-row FETCH statement. A row storage area can be a character variable with enough bytes to hold all of the rows that are requested on the multiple-row FETCH statement.

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.

Consider the following PL/I example:

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 296.

```
   *....+....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';
...

 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:
```

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 the 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.

Related reference

"DB2 for i5/OS sample tables" on page 277
These sample tables are referred to and used in the SQL programming and the SQL reference topic collections.

SQLDA (SQL descriptor area)

Unit of work and open cursors

When your program completes a unit of work, it should commit or roll back the changes that you have 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 Create SQL (CRTSQL*xxx*) commands can change the restoration of the cursor position for read-only cursors. For information about the use of the ALWBLK parameter and other performance-related options on the CRTSQL*xxx* commands, see "Dynamic SQL applications."

Related concepts

Commitment control

Dynamic SQL applications

Dynamic SQL allows an application to define and run SQL statements at program run time. An application that uses dynamic SQL either accepts an SQL statement as input or builds an SQL statement in the form of a character string. The application does not need to know the type of the SQL statement.

The application:
- Builds or accepts as input an SQL statement
- Prepares the SQL statement for running
- Runs the statement
- Handles SQL return codes

Notes:

- The processing of dynamic SQL can have substantially higher overhead than the processing of static SQL because the statement might need to be fully processed at run time. In the worst

case, the statement must be fully prepared, bound, and optimized by the database before it is run. In many other cases, if the statement has been run before, parts of the processing can be skipped because of the algorithms used and caches maintained by the database. These features allow the DB2 for i5/OS database to provide good performance for dynamic SQL statements. If performance for your dynamic application is critical, consider using the extended dynamic capability through the Process Extended Dynamic SQL (QSQPRCED) API. This feature allows the application to maintain a persistent cache of SQL statements and substantially reduces runtime overhead when the application runs.

- 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 the use of pointer variables. RPG/400 programs require the aid of PL/I, COBOL, C, or ILE RPG programs to manage the pointer variables.

Related concepts

"Using interactive SQL" on page 240
Interactive SQL allows a programmer or a database administrator to quickly and easily define, update, delete, or look at data for testing, problem analysis, and database maintenance.

Related reference

Actions allowed on SQL statements

Process Extended Dynamic SQL (QSQPRCED) API

"Unit of work and open cursors" on page 224
When your program completes a unit of work, it should commit or roll back the changes that you have made.

Designing and running a dynamic SQL application

To issue a dynamic SQL statement, use the statement with either an EXECUTE statement or an EXECUTE IMMEDIATE statement, because dynamic SQL statements are prepared at run time, not at precompile 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 Open Database Connectivity (ODBC) typically use dynamic SQL to access the database.

Related concepts

System i Access for Windows: Programming

CCSID of dynamic SQL statements

An SQL statement is normally a host variable. The coded character set identifier (CCSID) of the host variable 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. 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 represented by the value '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.

Processing non-SELECT statements

Before building a dynamic SQL non-SELECT statement, you need to verify that this SQL statement is allowed to be run dynamically.

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;
```

Related concepts

"Using interactive SQL" on page 240
Interactive SQL allows a programmer or a database administrator to quickly and easily define, update, delete, or look at data for testing, problem analysis, and database maintenance.

Using the PREPARE and EXECUTE statements:

If the non-SELECT statement does not contain parameter markers, you can run it dynamically using the EXECUTE IMMEDIATE statement. However, if the non-SELECT statement contains parameter markers, you must run it using the PREPARE and EXECUTE statements.

The PREPARE statement prepares the non-SELECT statement (for example, the DELETE statement) and gives it a statement name you choose. 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. 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 a descriptor

The basic types of SELECT statements are fixed list and varying list.

To process a fixed-list SELECT statement, an SQL descriptor is not necessary.

To process a varying-list SELECT statement, you must first declare an SQL descriptor area (SQLDA) structure or ALLOCATE an SQLDA. Both forms of SQL descriptors can be 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, a fixed-list SELECT statement retrieves data of a predictable number and type. When using this statement, you can anticipate and define host variables to accommodate the retrieved data so that an SQL descriptor area (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 in the same manner as 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 if you were using a varying-list SELECT statement.
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 need to use an SQL descriptor area.

Varying-list SELECT statements:

In dynamic SQL, a varying-list SELECT statement is used when the number and format of the result columns are not predictable; that is, you do not know the data types or the number of variables that you need.

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. REXX only supports SQL descriptors defined using the SQLDA structure; it does not support allocated SQL descriptors.

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. For an allocated SQL descriptor, run an ALLOCATE DESCRIPTOR statement to define the descriptor you intend to use.
6. 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 using an SQLDA and 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.

      If using an allocated SQL descriptor and the descriptor is not large enough, deallocate the descriptor, allocate it with a larger number of entries, and reissue the DESCRIBE statement.
7. For an SQLDA descriptor, allocate the amount of storage needed to contain a row of retrieved data.
8. For an SQLDA descriptor, put storage addresses into the SQLDA to tell SQL where to put each item of retrieved data.
9. FETCH a row.
10. Process the data returned in the SQL descriptor.
11. Handle any SQL return codes that might result.
12. When end of data occurs, close the cursor.
13. For an allocated SQL descriptor, run a DEALLOCATE DESCRIPTOR statement to delete the descriptor.

Related reference

"Example: A SELECT statement for allocating storage for SQLDA" on page 232
Suppose that your application needs to handle a dynamic SELECT statement that changes from one use to the next. This statement can be read from a display, passed in from another application, or built dynamically by your application.

SQL descriptor areas:

Dynamic SQL uses an SQL descriptor area to pass information about an SQL statement between SQL and your application.

A descriptor is required for running the DESCRIBE, DESCRIBE INPUT and DESCRIBE TABLE statements, and can also be used on the PREPARE, OPEN, FETCH, CALL, and EXECUTE statements.

The meaning of the information in a descriptor depends on its use. In PREPARE and DESCRIBE, a descriptor provides information to an application program about a prepared statement. In DESCRIBE INPUT, the SQL descriptor area provides information to an application program about parameter markers in a prepared statement. In DESCRIBE TABLE, the descriptor provides information to an application program about the columns in a table or view. In OPEN, EXECUTE, CALL, and FETCH, a descriptor provides information about host variables. For example, you can read values into the descriptor by using a DESCRIBE statement, change the data values in the descriptor to use the host variables, and then reuse the same descriptor in a FETCH statement.

If your application allows you to have several cursors open at the same time, you can code several descriptors, one for each dynamic SELECT statement.

There are two types of descriptors. One is defined with the ALLOCATE DESCRIPTOR statement. The other is defined with the SQLDA structure.

ALLOCATE DESCRIPTOR is not supported in REXX. SQLDAs can be used in C, C++, COBOL, PL/I, REXX, and RPG. Because RPG/400 does not provide a way to set pointers, the SQLDA must be set outside the RPG/400 program by a PL/I, C, C++, COBOL, or an ILE RPG program. That program must then call the RPG/400 program.

> **Related reference**
>
> SQLCA (SQL communication area)
>
> SQLDA (SQL descriptor area)

SQLDA format:

An SQL descriptor area (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.

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 before it is used 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 before being used 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 before being used 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 before being used. These variables are defined as follows:

SQLTYPE
SQLTYPE is a 2-byte integer that specifies the data type of the host variable or column. See SQLTYPE and SQLLEN for a table of the valid values. 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 i5/OS, 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. See CCSID values in SQLDATA or SQLNAME for the layout of the CCSID data in this field.

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.

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.

Related tasks

Coding SQL statements in REXX applications

Related reference

"Example: A SELECT statement for allocating storage for SQLDA"
Suppose that your application needs to handle a dynamic SELECT statement that changes from one use to the next. This statement can be read from a display, passed in from another application, or built dynamically by your application.

Example: A SELECT statement for allocating storage for SQLDA:

Suppose that your application needs to handle a dynamic SELECT statement that changes from one use to the next. This statement can be read from a display, passed in from another application, or built dynamically by your application.

In other words, you don't know exactly what this statement is going to be returning every time. Your application needs 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. Because this SELECT statement is 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.

Table 43. 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 44. SQLVAR element 1

Description	Value
SQLTYPE	453
SQLLEN	3
SQLDATA (3:4)	37
SQLNAME	8 WORKDEPT

Table 45. SQLVAR element 2

Description	Value
SQLTYPE	453
SQLLEN	4
SQLDATA(3:4)	37
SQLNAME	7 PHONENO

Your program might need 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.

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 SQLLEN to the length of the column. Because the data type of WORKDEPT is CHAR, SQL sets SQLLEN 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 typically 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

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 46. SQLDA header

Description	Value
SQLAID	'SQLDA'
SQLDABC	1616
SQLN	20
SQLD	2

Table 47. 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 48. 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.

Related reference

"Varying-list SELECT statements" on page 228
In dynamic SQL, a varying-list SELECT statement is used when the number and format of the result columns are not predictable; that is, you do not know the data types or the number of variables that you need.

"SQLDA format" on page 229
An SQL descriptor area (SQLDA) consists of four variables followed by an arbitrary number of occurrences of a sequence of six variables collectively named SQLVAR.

Example: A SELECT statement using an allocated SQL descriptor:

Suppose that your application needs to handle a dynamic SELECT statement that changes from one use to the next. This statement can be read from a display, passed from another application, or built dynamically by your application.

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 to allocate the largest number of entries for an SQL descriptor that you think you will need. Assume that no more than 20 columns are ever expected to be accessed by a single SELECT statement.
```
EXEC SQL
ALLOCATE DESCRIPTOR 'mydescr' WITH MAX 20;
```

Now that the descriptor is allocated, the DESCRIBE statement can be done to get the column information.
```
EXEC SQL
DESCRIBE S1 USING DESCRIPTOR 'mydescr';
```

When the DESCRIBE statement is run, SQL places values that provide information about the statement's select-list into the SQL descriptor area defined by 'mydescr'.

If the DESCRIBE determines that not enough entries were allocated in the descriptor, SQLCODE +239 is issued. As part of this diagnostic, the second replacement text value indicates the number of entries that are needed. The following code sample shows how this condition can be detected and shows the descriptor allocated with the larger size.

```
/* Determine the returned SQLCODE from the DESCRIBE statement */
EXEC SQL
  GET DIAGNOSTICS CONDITION 1: returned_sqlcode = DB2_RETURNED_SQLCODE;

if returned_sqlcode = 239 then do;

/* Get the second token for the SQLCODE that indicated
     not enough entries were allocated */

EXEC SQL
    GET DIAGNOSTICS CONDITION 1: token = DB2_ORDINAL_TOKEN_2;
  /* Move the token variable from a character host variable into an integer host variable */
  EXEC SQL
    SET :var1 = :token;
  /* Deallocate the descriptor that is too small */
  EXEC SQL
    DEALLOCATE DESCRIPTOR 'mydescr';
  /* Allocate the new descriptor to be the size indicated by the retrieved token */
  EXEC SQL
    ALLOCATE DESCRIPTOR 'mydescr' WITH MAX :var1;
  /* Perform the describe with the larger descriptor */
  EXEC SQL
    DESCRIBE s1 USING DESCRIPTOR 'mydescr';
end;
```

At this point, the descriptor contains the information about the SELECT statement. Now you are ready to retrieve the SELECT statement results. For dynamic SQL, the SELECT INTO statement is not allowed. You must use a cursor.

```
EXEC SQL
  DECLARE C1 CURSOR FOR S1;
```

You will notice that the prepared statement name is used in the cursor declaration instead of the complete SELECT statement. Now you can loop through the selected rows, processing them as you read them. The following code sample shows how this is done.

```
EXEC SQL
  OPEN C1;

EXEC SQL
    FETCH C1 USING SQL DESCRIPTOR 'mydescr';
do while not at end of data;

 /* process current data returned  (see below for discussion of doing this) */

/* then read the next row */

 EXEC SQL
    FETCH C1 USING SQL DESCRIPTOR 'mydescr';
end;

EXEC SQL
  CLOSE C1;
```

The cursor is opened. The result rows from the SELECT statement 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 the descriptor area.

After the FETCH has been processed, you can use the GET DESCRIPTOR statement to read the values. First, you must read the header value that indicates how many descriptor entries were used.

```
EXEC SQL
  GET DESCRIPTOR 'mydescr' :count = COUNT;
```

Next you can read information about each of the descriptor entries. After you determine the data type of the result column, you can do another GET DESCRIPTOR to return the actual value. To get the value of the indicator, specify the INDICATOR item. If the value of the INDICATOR item is negative, the value of the DATA item is not defined. Until another FETCH is done, the descriptor items will maintain their values.

```
do i = 1 to count;
  GET DESCRIPTOR 'mydescr' VALUE :i  /* set entry number to get */
                          :type = TYPE,                 /* get the data type */
                          :length = LENGTH,             /* length value */
                          :result_ind = INDICATOR;
 if result_ind >= 0 then
    if type = character
     GET DESCRIPTOR 'mydescr' VALUE :i
                        :char_result = DATA;      /* read data into character field */
    else
    if type = integer
       GET DESCRIPTOR 'mydescr' VALUE :i
                        :int_result = DATA;       /* read data into integer field */
  else
      /* continue checking and processing for all data types that might be returned */
end;
```

There are several other descriptor items that you might need to check to determine how to handle the result data. PRECISION, SCALE, DB2_CCSID, and DATETIME_INTERVAL_CODE are among them. The host variable that has the DATA value read into it must have the same data type and CCSID as the data being read. If the data type is varying length, the host variable can be declared longer than the actual data. For all other data types, the length must match exactly.

NAME, DB2_SYSTEM_COLUMN_NAME, and DB2_LABEL can be used to get name-related values for the result column. See GET DESCRIPTOR for more information about the items returned for a GET DESCRIPTOR statement and for the definition of the TYPE values

Parameter markers:

A *parameter marker* is a question mark (?) that appears in a dynamic statement string. The question mark can appear where a host variable might appear if the statement string were a static SQL statement.

In the example used, the SELECT statement that was dynamically run had a constant value in the WHERE clause:

```
    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 looks like this:

```
    SELECT WORKDEPT, PHONENO
        FROM CORPDATA.EMPLOYEE
        WHERE LASTNAME = ?
```

When using parameter markers, your application does not need to set the data types and values for the parameters until run time. By specifying a descriptor on the OPEN statement, you can substitute the values for the parameter markers in the SELECT statement.

To code such a program, you need to use the OPEN statement with a 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 descriptor entry. The descriptor name that you specify with this statement must identify a descriptor that contains a valid definition of the values. This descriptor is not used to return information about data items that are part of a SELECT list. It provides information about values that are used to replace parameter markers in the SELECT statement. It gets this information from the application, which must be designed to place appropriate values into the fields of the descriptor. The descriptor is then ready to be used by SQL for replacing parameter markers with the actual values.

When you use an SQLDA for input to the OPEN statement with the USING DESCRIPTOR clause, not all of its fields need 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 values, you need to determine:

- How many parameter markers there are
- The data types and attributes of these parameters markers (SQLTYPE, SQLLEN, and SQLNAME)
- Whether an indicator variable is needed

In addition, if the routine is to handle both SELECT and non-SELECT statements, you might want to determine what category of statement it is.

If your application uses parameter markers, your program has to perform the following steps. This can be done using either an SQLDA or an allocated descriptor.
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 or use ALLOCATE DESCRIPTOR to allocate a descriptor with that number of entries. This is not applicable in REXX.
4. For an SQLDA, set SQLN and SQLD to the number of parameter markers. SQLN is not applicable in REXX. For an allocated descriptor, use SET DESCRIPTOR to set the COUNT entry to the number of parameter markers.
5. For an SQLDA, 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. For an SQLDA, set SQLTYPE and SQLLEN for each parameter marker. For an allocated descriptor, use SET DESCRIPTOR to set the entries for TYPE, LENGTH, PRECISION, and SCALE for each parameter marker.
   c. For an SQLDA, allocate storage to hold the input values.
   d. For an SQLDA, set these values in storage.
   e. For an SQLDA, set SQLDATA and SQLIND (if applicable) for each parameter marker. For an allocated descriptor, use SET DESCRIPTOR to set entries for DATA and INDICATOR (if applicable) for each parameter marker.
   f. If character variables are used and they have a CCSID other than the job default CCSID, or graphic variables are used and they have a CCSID other than the associated DBCS CCSID for the job CCSID,
      - For an SQLDA, set SQLNAME (SQLCCSID in REXX) accordingly.
      - For an allocated SQL descriptor, use SET DESCRIPTOR to set the DB2_CCSID value.
   g. Issue the OPEN statement with a USING DESCRIPTOR clause (for an SQLDA) or USING SQL DESCRIPTOR clause (for an allocated descriptor) to open your cursor and substitute values for each of the parameter markers.

The statement can then be processed normally.

Related reference

Suppose that your application needs to handle a dynamic SELECT statement that changes from one use to the next. This statement can be read from a display, passed in from another application, or built dynamically by your application.

Suppose that your application needs to handle a dynamic SELECT statement that changes from one use to the next. This statement can be read from a display, passed from another application, or built dynamically by your application.

Using interactive SQL

Interactive SQL allows a programmer or a 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 about 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.
  - Start the 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 supported 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.

  Related concepts

  "Dynamic SQL applications" on page 224
  Dynamic SQL allows an application to define and run SQL statements at program run time. An application that uses dynamic SQL either accepts an SQL statement as input or builds an SQL statement in the form of a character string. The application does not need to know the type of the SQL statement.

Related reference

"Processing non-SELECT statements" on page 226
Before building a dynamic SQL non-SELECT statement, you need to verify that this SQL statement is allowed to be run dynamically.

Starting interactive SQL

To start using interactive SQL, enter STRSQL from an i5/OS command line.

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

```
                         Enter SQL Statements

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

Press F24=More keys to view the remaining function keys.

```
     _____
     _____
     _____
     _____
                                            Bottom

 F14=Delete line    F15=Split line    F16=Select collections (libraries)
 F17=Select tables                    F18=Select columns    F24=More keys
         (files)                              (fields)
```

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 workstation ID. This session-ID concept allows multiple users with the same user ID to use interactive SQL from more than one workstation at the same time. Also, more than one interactive SQL session can be run from the same workstation 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.

> **Related reference**
>
> Start SQL Interactive Session (STRSQL) command

Using the statement entry function

The statement entry function is the function that 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.

The statement you type on the command line can be one or more lines long. You can type bracketed comments (/* */) in interactive SQL. However, you should not use simple comments (that is, comments starting with --) in interactive SQL because these comments then include the remainder of the SQL statement within the comment. 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. If you press F9=Retrieve with your cursor on the statement entry line, your previous statement is copied to the input area. Pressing F9 again causes it to scroll back one more statement and copy that to the input area. Continuing to press F9 allows you to scroll back through your previous statements until you find the one that you want. If you need more room to type an SQL statement, page down on the display.

Prompting

The prompt function helps you provide the necessary information for the syntax of the statement that you want to use. The prompt function can be used in any of these statement processing modes: *RUN, *VLD, and *SYN. Prompting is not available for all SQL statements and is not complete for many SQL statements.

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:

```
                     Select SQL Statement

 Select one of the following:

      1. ALTER TABLE
      2. CALL
      3. COMMENT ON
      4. COMMIT
      5. CONNECT
      6. CREATE ALIAS
      7. CREATE COLLECTION
      8. CREATE INDEX
      9. CREATE PROCEDURE
     10. CREATE TABLE
     11. CREATE VIEW
     12. DELETE
     13. DISCONNECT
     14. DROP ALIAS
                                                      More...
 Selection
     __

 F3=Exit   F12=Cancel
```

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.

CREATE TABLE prompting:

You can enter column definitions individually when you are prompted for a CREATE TABLE statement.

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 double-byte character set (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 looks like this on the Enter SQL Statements display:

SELECT * **FROM** TABLE1 **WHERE** COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSS>'

Using the list selection function

You can access the list selection function by pressing F4 (Prompt) on certain prompt displays. To access the function on the Enter SQL Statements display, press F16 (Select collections), F17 (Select tables), or F18 (Select columns).

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. Then, from the column list, press F17 to change the table. If the table list were selected first, the table name will be inserted into your statement. You do 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.

> **Related reference**
>
> "Starting interactive SQL" on page 241
> To start using interactive SQL, enter STRSQL from an i5/OS command line.

Example: Using the list selection function:

This example shows how to use the list selection function to build a SELECT statement.

Assume you have:

- Just entered interactive SQL by typing STRSQL on an i5/OS 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

You can change session attributes from the Session Services display. You can also print, clear, or save the session to a source file.

To access the Session Services display, press F13 (Services) on the Enter SQL Statements display.

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.
- The SQL rules.
- The CONNECT password option.

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

To exit the interactive SQL environment, press F3 (Exit) on the Enter SQL Statements display. Several options are available for exiting.
- Save and exit session. Leave interactive SQL. Your current session will be saved and used the next time you start interactive SQL.
- Exit without saving session. Leave interactive SQL without saving your session.
- Resume session. Remain in interactive SQL and return to the Enter SQL Statements display. The current session parameters remain in effect.
- 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 selecting option 1 (Save and exit session) on the Exit Interactive SQL display, you can resume the 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 workstation. 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 you enter the Start SQL Interactive Session (STRSQL) command.

From this display, you can choose to do either of the following two things.
- 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 establishes an implicit connection to the local relational database when starting an SQL session. When the CONNECT statement is completed, a message shows the relational database connection that was established. If interactive SQL is starting a new session and COMMIT(*NONE) was not specified, or if interactive SQL is restoring a saved session and the commitment control level saved with the session was not *NONE, the connection will be registered with commitment control. This implicit connection and possible commitment control registration might influence subsequent connections to remote databases. It is suggested that you perform one of the following tasks before connecting to the remote database:

- When you are connecting to an application server that does not support distributed unit of work, issue a RELEASE ALL statement followed by a COMMIT statement to end previous connections, including the implicit connection to the local database.
- When you are connecting to an application server other than DB2 for i5/OS, issue a RELEASE ALL statement followed by a COMMIT statement to end previous connections, including the implicit connection to the local database, and change the commitment control level to at least *CHG.

When you are connecting to an application server other than DB2 for i5/OS, some session attributes are changed to attributes that are supported by that application server. The following table shows the attributes that change.

Table 49. Values table

Session attribute	Original value	New value
Date format		
	*YMD	*ISO
	*DMY	*EUR
	*MDY	*USA
	*JUL	*USA
Time format	*HMS with a : separator *HMS with any other separator	*JIS
		*EUR
Commitment control		*CS Repeatable Read
	*CHG,	
	*NONE	
	*ALL	
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

Note: When you are connected to a DB2 for Linux®, UNIX®, and Windows® or DB2 for z/OS® application server, the date and time formats specified must be the same.

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 application servers other than DB2 for i5/OS might require some setup.

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.

Related concepts

Distributed database programming

Related reference

"Determining the connection type" on page 268
When a remote SQL connection is established, it uses either an unprotected or a protected network connection.

Using the SQL statement processor

The SQL statement processor allows SQL statements to be run from a source member or a source stream file. The statements in the source can be run repeatedly or changed without compiling the source. This makes the setup of a database environment easier.

The SQL statement processor is available through the Run SQL Statements (RUNSQLSTM) command.

The following statements can be used with the SQL statement processor:
- ALTER FUNCTION
- ALTER PROCEDURE
- ALTER SEQUENCE
- ALTER TABLE
- CALL
- COMMENT ON
- COMMIT
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE

- DROP
- GRANT
- INSERT
- LABEL ON
- LOCK TABLE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT
- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT DEGREE
- SET ENCRYPTION PASSWORD
- SET PATH
- SET SCHEMA
- SET TRANSACTION
- UPDATE

In the source, statements do not begin with EXEC SQL. Each statement ends with a semicolon. For a source member, the default right margin is 80. If the record length of the source member is longer than 80, only the first 80 characters are read. You can change the right margin to some other value by using the MARGINS parameter on the RUNSQLSTM command. For a source stream file, the entire file is read; no margin is used.

Comments in the source 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, specify the PROCESS(*SYN) parameter on the RUNSQLSTM command. To see more details for error messages in the listing, specify the SECLVLTXT(*YES) parameter.

Related reference

Run SQL Statement (RUNSQLSTM) command

Execution of statements after errors occur

If a statement returns an error with a severity higher than the value specified for the error level (ERRLVL) parameter of the Run SQL Statements (RUNSQLSTM) command, the statement fails.

The rest of the statements in the source are parsed to check for syntax errors, and will not be run. 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. DROP statements issue a severity level 20 error if the object is not found to be dropped. Setting the ERRLVL parameter to have a value of 20 allows you to ignore these errors for DROP statements while not allowing processing to continue for other higher severity errors.

Commitment control in the SQL statement processor

A commitment-control level is specified on the Run SQL Statements (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 runs, 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.

Source listing for the SQL statement processor

This example shows an output source listing for the SQL statement processor.

Note: By using the code examples, you agree to the terms of "Code license and disclaimer information" on page 296

```
xxxxSS1 VxRxMx yymmdd        Run SQL Statements        SCHEMA      02/15/08 15:35:18   Page   1
Source file..............CORPDATA/SRC
Member...................SCHEMA
Commit...................*NONE
Naming...................*SYS
Generation level.........10
Date format..............*JOB
Date separator...........*JOB
Right margin.............80
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................37
Statement processing.....*RUN
Allow copy of data.......*OPTIMIZE
Allow blocking...........*ALLREAD
SQL rules................*DB2
Decimal result options:
  Maximum precision......31
  Maximum scale..........31
  Minimum divide scale...0
Source member changed on 11/01/07  11:54:10
```

Figure 1. QSYSPRT listing for SQL statement processor

```
xxxxSS1 VxRxMx yymmdd       Run SQL Statements       SCHEMA       02/15/08 15:35:18   Page   2
Record  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 SEQNBR  Last change
    1
    2      DROP SCHEMA DEPT;
    3      DROP SCHEMA MANAGER;
    4
    5      CREATE SCHEMA DEPT
    6          CREATE TABLE EMP (EMPNAME CHAR(50), EMPNBR INT)
    7                          -- EMP will be created in schema 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 2. QSYSPRT listing for SQL statement processor (continued)

```
xxxxSS1 VxRxMx yymmdd       Run SQL Statements       SCHEMA          02/15/08 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 Schema 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 Schema MANAGER created.
SQL7950   0     19   Position 9 Table EMP_SALARY created in schema
                     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 3. QSYSPRT listing for SQL statement processor (continued)

Distributed relational database function and SQL

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, the DB2 for i5/OS database) or of different types (DB2 for z/OS, DB2 for VSE and VM, or non-IBM database management systems that support Distributed Relational Database Architecture (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 for i5/OS provides support for DRDA to allow an application requester to communicate with application servers. In addition, DB2 for i5/OS can call exit programs to allow access to data on other database management systems that do not support DRDA. These exit programs are called *application requester driver (ARD) programs*.

DB2 for i5/OS supports the following levels of distributed relational database:
- Remote unit of work (RUW)

  A *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 for i5/OS supports RUW over either Advanced Program-to-Program Communication (APPC) or TCP/IP.
- Distributed unit of work (DUW)

  A *distributed unit of work* is where the preparation and running of SQL statements can occur at multiple application servers during a unit of work. However, a single SQL statement can only refer to objects located at a single application server. DB2 for i5/OS supports DUW over APPC and DUW over TCP/IP.

  Related concepts

  "Introduction to DB2 for i5/OS Structured Query Language" on page 3
  Structured Query Language (SQL) is a standardized language for defining and manipulating data in a relational database. These topics describe the System i implementation of the SQL using the DB2 for i5/OS database and the IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program.

  "SQL packages" on page 12
  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).

  Distributed database programming

DB2 for i5/OS distributed relational database support

The IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program supports interactive access to distributed databases.

DB2 Query Manager and SQL Development Kit for i5/OS provides distributed relational database support through the following SQL statements:
- CONNECT
- SET CONNECTION
- DISCONNECT
- RELEASE
- DROP PACKAGE
- GRANT PACKAGE
- REVOKE PACKAGE

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

Related tasks

Preparing and running a program with SQL statements

Related reference

"DB2 for i5/OS CL command descriptions" on page 296
DB2 for i5/OS provides these CL commands for SQL.

CONNECT (Type 1)

CONNECT (Type 2)

DISCONNECT

DROP

GRANT (Package Privileges)

REVOKE (Package Privileges)

RELEASE (Connection)

SET CONNECTION

DB2 for i5/OS distributed relational database example program

A relational database example program that uses remote unit of work is included with the SQL product.
Several files and members within the QSQL library can help you set up an environment that runs a
distributed DB2 for i5/OS example 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 example 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 example program, specify the following command on the command line:
```
========> ADDLIBLE QSQSAMP
```

To call the first display that allows you to customize the example program, specify the following
command on the command line.
```
========> CALL QSQ8HC3
```

The following display opens. From this display, you can customize your database example program.

```
SAMPLE 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
```

SQL package support

The i5/OS operating system supports SQL packages. The 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 the i5/OS operating system, the EXECUTE privilege includes system authority of *OBJOPR and *EXECUTE.

> **Related reference**
>
> Create SQL Package (CRTSQLPKG) command

Valid SQL statements in an SQL package

Programs that are connected to another server can use any of the SQL statements except the SET TRANSACTION statement.

Programs compiled with DB2 for i5/OS that refer to a remote database that is not DB2 for i5/OS can use executable SQL statements supported by the remote database. The precompiler continues to issue diagnostic messages for statements that it does not understand. These statements are sent to the remote system during the creation of the SQL package. The runtime support returns an 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 the application server are running on the OS/400® V5R2 or i5/OS V5R3, or later, with the following exception. An application requester that is not running on the i5/OS operating system can issue read-only, insensitive scrollable cursor operations on an i5/OS V5R3 application server. A further restriction on the use of multiple-row FETCH statements, blocked INSERT statements, and scrollable cursors is that the transmission of binary large object (BLOB), character large object (CLOB), and double-byte character large object (DBCLOB) data is not allowed when those functions are used.

Related information

Characteristics of SQL statements

Considerations for creating an SQL package

When you create an SQL package, consider these aspects.

CRTSQLPKG authorization:

When you create an SQL package on the i5/OS operating system, the authorization ID used must have *USE authority to the Create SQL Package (CRTSQLPKG) command.

Creating a package on a database other than DB2 for i5/OS:

When you create a program or an SQL package on a database other than DB2 for i5/OS, and try to use SQL statements that are unique to the other database, you must set the GENLVL parameter on the Create SQL (CRTSQL*xxx*) commands 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 generate a message severity greater than 30.

The precompiler listing should be checked for unexpected messages when running with a GENLVL value greater than 10. When you are creating a package for a DB2 product, you must set the GENLVL parameter to a value less than 20.

If the RDB parameter specifies a database that is not DB2 for i5/OS, the following options should not be used on the CRTSQL*xxx* command:
* COMMIT(*NONE)
* OPTION(*SYS)
* DATFMT(*MDY)
* DATFMT(*DMY)
* DATFMT(*JUL)
* DATFMT(*YMD)
* DATFMT(*JOB)
* DYNUSRPRF(*OWNER)
* TIMFMT(*HMS) if TIMSEP(*BLANK) or TIMSEP(',') is specified
* SRTSEQ(*JOBRUN)
* SRTSEQ(*LANGIDUNQ)
* SRTSEQ(*LANGIDSHR)
* SRTSEQ(library-name/table-name)

Note: When you connect to a DB2 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) parameter:

When you create 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 before this release. TGTRLS message are suppressed when the TGTRLS parameter is *CURRENT.

SQL statement size:

The create SQL package function might not be able to handle an SQL statement of the same size that the precompiler can process.

While the SQL program is being precompiled, 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 can 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 is not created and the program still runs. 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 used for the package creation is based on the type of connection specified by 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 is created on the local system. The SQL package can be saved (Save Object (SAVOBJ) command) and then restored (Restore Object (RSTOBJ) command) to another system. 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 an 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; otherwise, the package cannot be used. It is possible for the program and SQL package to appear to be uncoordinated. Assume that the program and the application server are on two distinct i5/OS operating systems. The program is running in session A and it is re-created in session B (where the SQL package is also re-created). The next call to the program in session A might cause 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. You can avoid this situation 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 start SQL from an attention key program while you are already precompiling, you will receive unpredictable results.

The Create SQL (CRTSQL*xxx*), Create SQL Package (CRTSQLPKG), and Start SQL Interactive Session (STRSQL) commands and the SQL runtime environment are not recursive. They produce unpredictable results if recursion is attempted. Recursion occurs 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 a System i product, the server on the System i platform cannot have the job coded character set identifier (CCSID) value 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, the SQL package name, the library name, and the text of the SQL package from the CCSID of the job to CCSID 500. This is required by Distributed Relational Database Architecture (DRDA). When the remote system is a System i product, 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 displays "a[b]c". If you created a program that referenced the table whose name was "a¬b|c.", the program will 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

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.

Before the use of TCP/IP by Distributed Relational Database Architecture (DRDA), the term *connection* was not ambiguous. It referred to a connection from the SQL point of view. That is, a connection starts when you run the CONNECT TO statement to connect to some relational database (RDB), and ends when you run the DISCONNECT or RELEASE ALL statement followed by a successful COMMIT operation. The Advanced Program-to-Program Communication (APPC) conversation might or might not have been kept up, depending on the DDMCNV attribute value of the job and whether the conversation was with a System i product or with other types of systems.

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* is replaced, in this topic, 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 might be the possibility of confusion between the two types of connections, the word will be qualified by SQL or network to help the reader to understand the intended meaning.

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

Here is the source code for PGM1.

```
...
EXEC SQL
  CONNECT TO SYSB
END-EXEC.
EXEC SQL
  SELECT ...
END-EXEC.
CALL PGM2.
...
```

Figure 4. Source code for PGM1

Command to create program and SQL package for PGM1:

```
CRTSQLCBL PGM(PGM1) COMMIT(*NONE) RDB(SYSB)
```

Source code for PGM2

Here is the 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 5. 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

Here is the source code for PGM3.

```
   ...
    EXEC SQL
      INSERT INTO TAB VALUES(:st1);
    EXEC SQL COMMIT;
   ...
```

Figure 6. 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)
```

SYSA
Job:
1 Call

Default Activation Group:

PGM1 **2** Connect

3 Call

System-Named Activation Group:

PGM2 **4** Connect

5 **8**
Call Return Call
7

Activation Group APPGRP:

PGM3 **6** Connect

SYSB (Remote)
Job:
Default Activation Group:
SQL Package for PGM1

SYSC (Remote)
Job:
Default Activation Group:
SQL Package for PGM2

SYSD (Remote)
Job:
Default Activation Group:
SQLPackage for PGM3
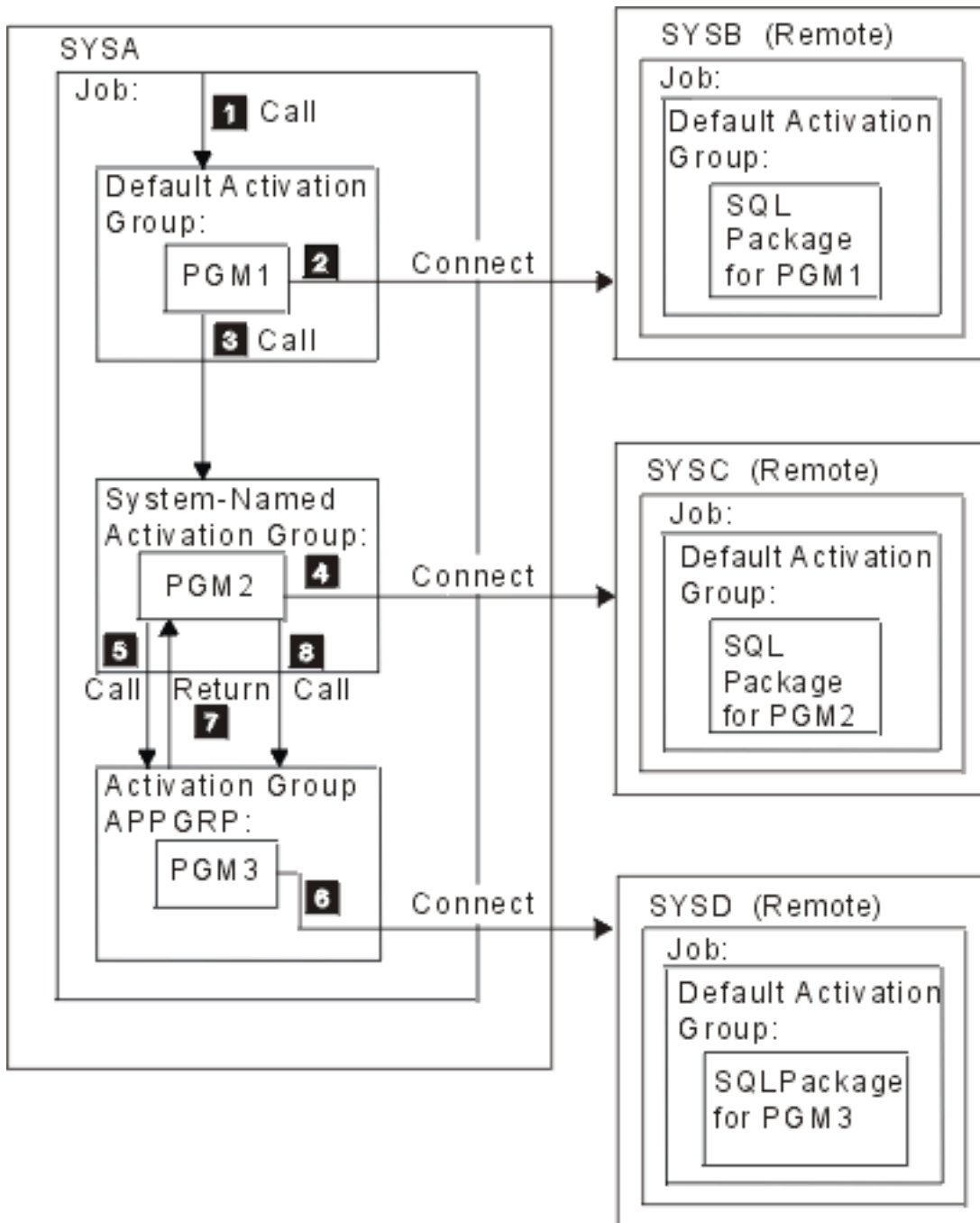
In this example, PGM1 is a non-ILE program created using the CRTSQLCBL command. This program runs in the default activation group. PGM2 is created using the CRTSQLCI command, and it runs in a system-named activation group. PGM3 is also created using the CRTSQLCI command, but it runs in the activation 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 are connected 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.

```
┌─────────────────────────────────┐   ┌─────────────────────────────────┐
│ SYSA                            │   │ SYSB                            │
│   Job:                          │   │   ┌───────────────────────────┐ │
│   ┌───────────────────────────┐ │   │   │ Job:                      │ │
│   │                           │ │   │   │                           │ │
│   │ ┌───────────────────────┐ │ │   │   │                           │ │
│   │ │ Default Activation    │ │ │   │   │                           │ │
│   │ │ Group:                │ │ │   │   │                           │ │
│   │ │          Connect      │─┼─┼─┼───┼──►│                           │ │
│   │ │                       │ │ │   │   │                           │ │
│   │ └───────────────────────┘ │ │   │   └───────────────────────────┘ │
│   │                           │ │   │                                 │
│   │                           │ │   │   ┌───────────────────────────┐ │
│   │                           │ │   │   │ Job:                      │ │
│   │ ┌───────────────────────┐ │ │   │   │                           │ │
│   │ │ System-Named          │ │ │   │   │                           │ │
│   │ │ Activation Group:     │ Connect │ │                           │ │
│   │ │                       │─┼─┼─┼───┼──►│                           │ │
│   │ │                       │ │ │   │   │                           │ │
│   │ └───────────────────────┘ │ │   │   └───────────────────────────┘ │
│   │                           │ │   │                                 │
│   └───────────────────────────┘ │   │                                 │
└─────────────────────────────────┘   └─────────────────────────────────┘
```
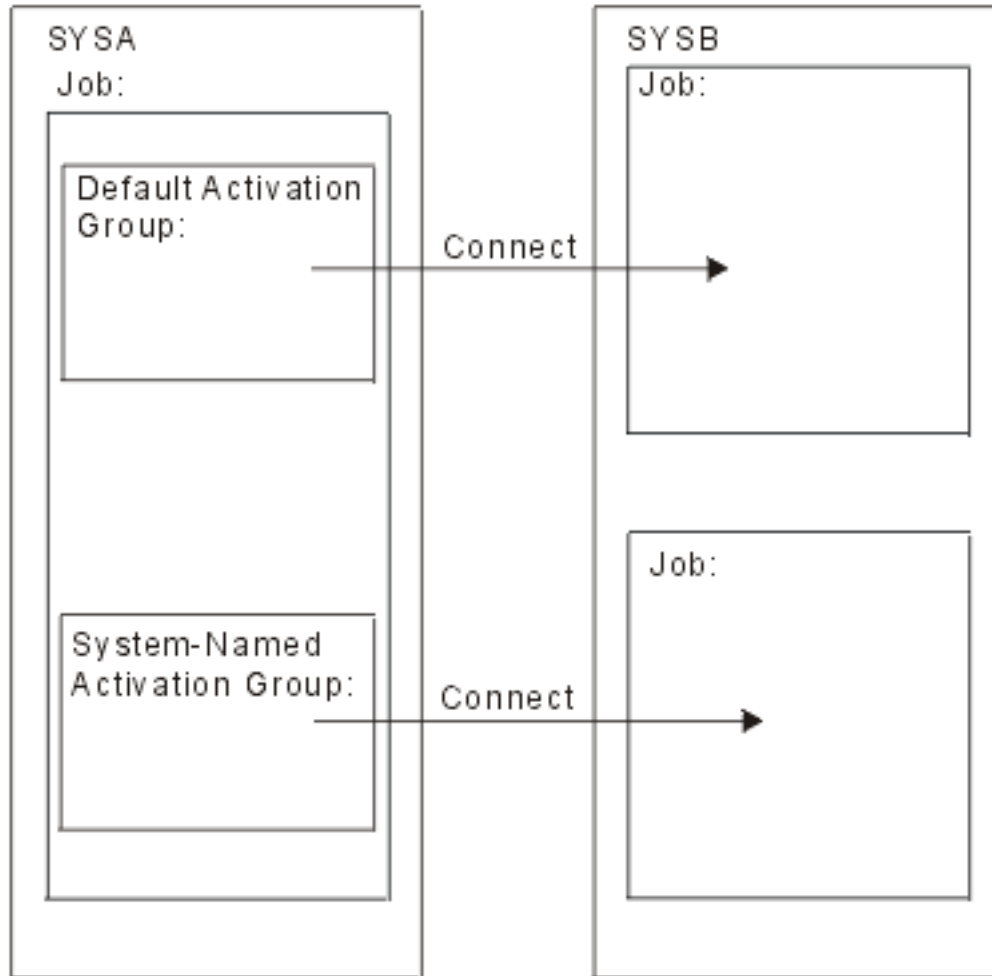
Implicit connection management for the default activation group

An application requester can be implicitly connected 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.

    Related reference

    "Ending connections" on page 272
    Because remote SQL connections use resources, you need to end the connections that are no longer used, as soon as possible. You can end connections implicitly or explicitly.

Implicit connection management for nondefault activation groups

An application requester can be implicitly connected 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.

    Related reference

    "Ending connections" on page 272
    Because remote SQL connections use resources, you need to end the connections that are no longer used, as soon as possible. You can end connections implicitly or explicitly.

Distributed support

DB2 for i5/OS supports these 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.

Determining the connection type

When a remote SQL connection is established, it uses either an unprotected or a protected network connection.

With regards to committable updates, this SQL connection may be read-only, updatable, or unknown whether it is updatable 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 and DB2_CONNECTION_TYPE of the SQL diagnostic area indicate the type of connection.

DB2_CONNECTION_TYPE specific values are:
1. The connection is to the local relational database and the connection is protected.
2. The connection is to a remote relational database and the connection is unprotected.
3. The connection is to a remote relational database and the connection is protected.
4. The connection is to an application requester driver program and the connection is protected.

SQLERRD(4) specific values are:
1. The connection is to a remote relational database and the connection is unprotected. Committable updates can be performed on the connection. This will occur when any of the following are true:
   - The connection is established using remote unit of work (RUW).
   - If the connection is established using distributed unit of work (DUW) then all the following are true:
     - The connection is not local.
     - The application server does not support distributed unit of work.
     - 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 updatable local files under commitment control for the commitment definition.
     - There are no open updatable 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 updatable DRDA resource for remote connections or a two-phase updatable DRDA resource for local and ARD connections.
2. The connection is to a remote relational database and the connection is unprotected. The connection is read-only. This will occur only when the following are true:
   - 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 updatable local files under commitment control for the commitment definition.
- There are open updatable 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 DRDA read-only resource.

3. The connection is to a remote relational database and the connection is protected. It is unknown if committable updates can be performed. This will occur when all of the following are true:
   - 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 DRDA undetermined resource.

4. The connection is to a remote relational database and the connection is unprotected. It is unknown if committable updates can be performed. This will occur only when all of the following are true:
   - 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. The connection is to the local database or an application requester driver (ARD) program and the connection is protected. It is unknown if committable updates can be performed. If running with commitment control, SQL will register a two-phase DRDA undetermined resource.

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 50. Summary of connection type

Connect under commitment control	Application server supports two-phase commit	Application server supports distributed unit of work	Other updatable 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
No	Yes	Yes	Yes	4
Yes	No	No	No	1
Yes	No	No	Yes	2
Yes	No	Yes	No	4

Table 50. Summary of connection type (continued)

Connect under commitment control	Application server supports two-phase commit	Application server supports distributed unit of work	Other updatable one-phase resource registered	SQLERRD(4)
Yes	No	Yes	Yes	4
Yes	Yes	No	No	N/A [1]
Yes	Yes	No	Yes	N/A [1]
Yes	Yes	Yes	No	3
Yes	Yes	Yes	Yes	3

[1]DRDA does not allow protected connections to be used to application servers that support only remote unit of work (DRDA1). This includes all DB2 for i5/OS TCP/IP connections.

Related concepts

Commitment control

Related reference

"Accessing remote databases with interactive SQL" on page 249
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.

Connect and commitment control restrictions

There are restrictions on when you can establish the connection using commitment control. The restrictions also apply if you attempt to run statements using commitment control, but you specified COMMIT(*NONE) on the connection.

If a two-phase undetermined or updatable resource is registered or a one-phase updatable resource is registered, another one-phase updatable 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 the connection status

A CONNECT statement without parameters can be used to determine whether the current connection is updatable or read-only for the current unit of work.

A value of 1 or 2 is returned in SQLERRD(3) in the SQLCA or DB2_CONNECTION_STATUS in the SQL diagnostic area. The value 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:

   * The connection is established using remote unit of work (RUW).
   * If the connection is established using distributed unit of work (DUW) and all of the following are true:

- 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 are open updatable local files under commitment control. .
  - There are open updatable DDM files that use protected connections.
  - There are two-phase API commitment control resources.
  - No committable updates have been made.
- If the connection is established using distributed unit of work (DUW) and all of the following are true:
  - 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 updatable DDM files that use protected connections.
  - There are no open updatable 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:
   - If the connection is established using distributed unit of work (DUW) and one of the following are true:
     - A connection exists to an updatable application server that only supports remote unit of work.
     - The first committable update is performed on a connection that uses an unprotected connection.
   - If the connection is established using distributed unit of work (DUW) and one of the following are true:
     - A connection exists to an updatable application server that only supports remote unit of work.
     - The first committable update was not performed on this connection.
     - There are open updatable DDM files that use protected connections.
     - There are open updatable local files under commitment control.
     - There are two-phase API commitment control resources.

The following table summarizes how the connection status is determined based on the connection type value, if there is an updatable connection to an application server that only supports remote unit of work, and where the first committable update occurred.

Table 51. Summary of determining connection status values

Connection method	Connection exists to updatable remote unit of work application server	Where first committable update occurred [1]	SQLERRD(3) or DB2_CONNECTION_STATUS
RUW			1
DUW	Yes		2
DUW	No	no updates	1
DUW	No	one-phase	2
DUW	No	this connection	1
DUW	No	two-phase	1

Table 51. Summary of determining connection status values (continued)

Connection method	Connection exists to updatable remote unit of work application server	Where first committable update occurred [1]	SQLERRD(3) or DB2_CONNECTION_STATUS

[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.

If an attempt is made to perform a committable update over a read-only 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 you connect in a distributed unit of work application, consider these points.

- 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 updatable and read-only status of a connection.
- If you connect using commitment control to an application server that does not support distributed unit of work (for example, a V4R5 system using TCP/IP), that connection is either updatable or read-only. If the connection is updatable, it is the only updatable connection. Since V5R3, Distributed Relational Database Architecture (DRDA) two-phase-commit operations have taken the following updates into consideration:
  - Updates done as a result of triggers
  - Updates done as a result of user-defined functions that are activated during a database query

Ending connections

Because remote SQL connections use resources, you need to end the connections that are no longer used, as soon as possible. You can end connections implicitly or explicitly.

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 stated 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 distributed data management (DDM) network connection (Advanced Program-to-Program Communication (APPC) conversation or TCP/IP connection) is used. When the SQL connection is ended, the network connection might 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 a server on the System i platform, the connection becomes unused. If the job attribute value is *DROP and the connection

is to a server on the System i platform, the connection is dropped. If the connection is to a server on a non-System i platform, 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.

> **Related reference**
>
> "Implicit connection management for the default activation group" on page 266
> An application requester can be implicitly connected to an application server.
>
> "Implicit connection management for nondefault activation groups" on page 267
> An application requester can be implicitly connected 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.

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 application servers to be committed or rolled back within a single unit of work.

Managing distributed unit of work connections

You can use the CONNECT, SET CONNECTION, DISCONNECT, and RELEASE statements to manage connections in the distributed unit of work (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;

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;
         }
      /* 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:

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 7. Example of distributed unit of work program

In this program, there are three application servers active: LOCALSYS which the local system, and two 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 cannot 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 the connection status

When it is possible to have a read-only connection, your program should check the status of the connection before doing any committable updates. This prevents 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 updatable.
      EXEC SQL CONNECT END-EXEC.
 * If connection is updatable, 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 8. 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 9. Example of cursors in a DUW program

Application requester driver programs

To complement database access provided by products that implement Distributed Relational Database Architecture (DRDA), DB2 for i5/OS provides an interface for writing exit programs on a DB2 for i5/OS application requester to process SQL requests. Such an exit program is called an *application requester driver (ARD)*.

The system 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).

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 about errors that are detected in the distributed data management (DDM) component of the i5/OS operating system from which an Authorized Program Analysis Report (APAR) can be created. By means of this function, key structures

and the DDM data stream are automatically dumped to a spooled file. The first 1024 bytes of the error information are also logged in the system error log. This automatic dumping of error information about the first occurrence of an error means that the failure does not need to be re-created to be reported by the customer. FFDC is active in both the application requester and application server functions of the i5/OS 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 *DRDA: Problem Determination Guide*, SC26-4782. This manual is available from the

IBM Publications Center➡ as a printed hardcopy that you can order or in an online format that you can download at no charge.

When an SQL error is detected, an SQLCODE with a corresponding SQLSTATE is returned in the SQLCA.

Related reference

SQL messages and codes

DRDA stored procedure considerations

The i5/OS Distributed Relational Database Architecture (DRDA) server supports the return of one or more result sets in a stored procedure.

You can generate result sets in a stored procedure by opening one or more SQL cursors associated with SELECT statements. In addition, a maximum of one array result set can also be returned. Before V5R3, only one instance of a query opened in a stored procedure was allowed to be open at one time. Now you can make multiple calls to the same stored procedure without closing the result set cursors so that more than one instance of a query can be opened simultaneously.

Related concepts

"Stored procedures" on page 122
A *procedure* (often called a stored procedure) is a program that can be called to perform operations. A procedure can include both host language statements and SQL statements. Procedures in SQL provide the same benefits as procedures in a host language.

Distributed database programming

Related reference

SET RESULT SETS

CREATE PROCEDURE (SQL)

CREATE PROCEDURE (External)

Reference

Reference information for SQL programming includes sample tables and CL commands.

DB2 for i5/OS sample tables

These sample tables are referred to and used in the SQL programming and the SQL reference topic collections.

Along with the tables are the SQL statements for creating the tables.

As a group, the tables include information that describes employees, departments, projects, and activities. This information forms a sample application that demonstrates some of the features of the IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program. All examples assume that the tables are in a schema named CORPDATA (for corporate data).

A stored procedure is included as part of the system that contains the data definition language (DDL) statements to create all of these tables and the INSERT statements to populate them. The procedure creates the schema specified on the call to the procedure. Because this is an external stored procedure, it can be called from any SQL interface, including interactive SQL and System i Navigator. To call the procedure where SAMPLE is the schema that you want 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.

Note: In these sample tables, a question mark (?) indicates a null value.

Related reference

"Referential integrity and tables" on page 17
Referential integrity is the condition of a set of tables in a database in which all references from one table to another are valid.

"Multiple-row FETCH using a row storage area" on page 222
Before using a multiple-row FETCH statement with the row storage area, the application must define a row storage area and an associated description area.

Department table (DEPARTMENT)

The department table describes each department in the enterprise and identifies its manager and the department that 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 52. 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.

DEPARTMENT:

Here is a complete listing of the data in the DEPARTMENT table.

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
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	?
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 every employee 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	Family 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

EMPLOYEE:

Here is a complete listing of the data in the EMPLOYEE table.

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 of each employee identified by an 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

EMP_PHOTO:

Here is a complete listing of the data in the EMP_PHOTO table.

EMPNO	PHOTO_FORMAT	PICTURE	EMP_ROWID
000130	bitmap	?	
000130	gif	?	
000140	bitmap	?	
000140	gif	?	
000150	bitmap	?	
000150	gif	?	
000190	bitmap	?	
000190	gif	?	

Employee resumé table (EMP_RESUME)

The employee resumé table contains a resumé for each employee identified by an employee number.

The employee resumé table 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	Resumé

Column name	Description
EMP_ROWID	Unique row id, not currently used

EMP_RESUME:

Here is a complete listing of the data in the EMP_RESUME table.

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 (EMPPROJACT)

The employee to project activity table identifies the employee who participates in each activity listed for each project. The employee's level of involvement (full-time or part-time) and the activity schedule are also included in the table.

The employee to project activity table is created with the following CREATE TABLE and ALTER TABLE statements:

```
CREATE TABLE EMPPROJACT
     (EMPNO    CHAR(6)       NOT NULL,
      PROJNO   CHAR(6)       NOT NULL,
      ACTNO    SMALLINT      NOT NULL,
      EMPTIME  DECIMAL(5,2)          ,
      EMSTDATE DATE                  ,
      EMENDATE DATE                  )

ALTER TABLE EMPPROJACT
             ADD FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
             REFERENCES PROJACT
             ON DELETE RESTRICT
```

The following aliases are created for the table:

```
CREATE ALIAS EMPACT FOR EMPPROJACT
```

```
CREATE ALIAS EMP_ACT FOR EMPPROJACT
```

The table below shows the content of the columns.

Table 53. Columns of the Employee to project activity table

Column name	Description
EMPNO	Employee ID number
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

Table 53. Columns of the Employee to project activity table (continued)

Column name	Description
EMSTDATE	Start date of the activity
EMENDATE	Completion date of the activity

EMPPROJACT:

Here is a complete listing of the data in the EMPPROJACT table.

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
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

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
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
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
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

PROJECT:

Here is a complete listing of the data in the PROJECT table.

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 (PROJACT)

The project activity table describes each project activity that the business is currently undertaking. Data contained in each row includes 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 PROJACT
          (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 PROJACT
          ADD FOREIGN KEY RPAP (PROJNO)
          REFERENCES PROJECT
          ON DELETE RESTRICT
```

The following foreign key is added later:

```
ALTER TABLE PROJACT
          ADD FOREIGN KEY RPAA (ACTNO)
             REFERENCES ACT
             ON DELETE RESTRICT
```

The following index is created:

```
CREATE UNIQUE INDEX XPROJAC1
          ON PROJACT (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

PROJACT:

Here is a complete listing of the data in the PROJACT table.

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	?
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	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
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	?
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	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
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

ACT:

Here is a complete listing of the data in the ACT table.

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

ACTNO	ACTKWD	ACTDESC
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

CL_SCHED:

Here is a complete listing of the data in the CL_SCHED table.

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 that contains the timestamp when a message is received, the user ID of the person who sent the message, and the content of the message.

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

IN_TRAY:

Here is a complete listing of the data in the IN_TRAY table.

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
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

ORG:

Here is a complete listing of the data in the ORG table.

DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
10	Head Office	160	Corporate	New York
15	New England	50	Eastern	Boston
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 background information about 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

Column name	Description
SALARY	Employee's annual salary
COMM	Employee's commission

STAFF:

Here is a complete listing of the data in the STAFF table.

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	?
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

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
3650	Gafney	84	Clerk	5	13030.50	188.00

Sales table (SALES)

The sales table describes the information about each sale 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

SALES:

Here is a complete listing of the data in the SALES table.

SALES_DATE	SALES_PERSON	REGION	SALES
12/31/1995	LUCCHESSI	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	LUCCHESSI	Ontario-South	3
03/29/1996	LUCCHESSI	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	LUCCHESSI	Ontario-South	1
03/30/1996	LUCCHESSI	Quebec	2
03/30/1996	LUCCHESSI	Manitoba	1
03/30/1996	LEE	Ontario-South	7
03/30/1996	LEE	Ontario-North	3
03/30/1996	LEE	Quebec	7

SALES_DATE	SALES_PERSON	REGION	SALES
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	LUCCHESSI	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	LUCCHESSI	Ontario-South	3
04/01/1996	LUCCHESSI	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
04/01/1996	GOUNOT	Ontario-North	1
04/01/1996	GOUNOT	Quebec	3
04/01/1996	GOUNOT	Manitoba	7

DB2 for i5/OS CL command descriptions

DB2 for i5/OS provides these CL commands for SQL.

- Create SQL Package (CRTSQLPKG) command
- Delete SQL Package (DLTSQLPKG) command
- Print SQL Information (PRTSQLINF) command
- Run SQL Statement (RUNSQLSTM) command
- Start SQL Interactive Session (STRSQL) command

   Related reference

   "DB2 for i5/OS distributed relational database support" on page 255
   The IBM DB2 Query Manager and SQL Development Kit for i5/OS licensed program supports
   interactive access to distributed databases.

Code license and disclaimer information

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;

2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR

3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This SQL programming publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

DB2
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
i5/OS
IBM
IBM (logo)
Integrated Language Environment
iSeries
Lotus Notes
OS/400
REXX
RPG/400
System i
System/36
WebSphere
z/OS

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM ®

Printed in USA