Chapter 2. Concepts

The chapter provides an overview of the concepts commonly used in the Structured Query Language (SQL). The intent of the chapter is to provide a high-level view of the concepts. The reference material that follows provides a more detailed view.

Relational Database

A *relational database* is a database that can be perceived as a set of tables and manipulated in accordance with the relational model of data. It contains a set of objects used to store, manage, and access data. Examples of such objects are tables, views, indexes, functions, triggers, and packages.

A *partitioned* relational database is a relational database where the data is managed across multiple partitions (also called nodes). This partitioning of data across partitions is transparent to users of most SQL statements. However, some DDL statements take partition information into consideration (e.g. CREATE NODEGROUP).

A *federated* database is a relational database where the data is stored in multiple data sources (such as separate relational databases). The data appears as if it were all in a single large database and can be accessed through traditional SQL queries. Changes to the data can be explicitly directed to the appropriate data source. See “DB2 Federated Systems” on page 41 for more information.

Structured Query Language (SQL)

SQL is a standardized language for defining and manipulating data in a relational database. In accordance with the relational model of data, the database is perceived as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more base tables.

SQL statements are executed by a database manager. One of the functions of the database manager is to transform the specification of a result table into a sequence of internal operations that optimize data retrieval. The transformation occurs in two phases: preparation and binding.

All executable SQL statements must be prepared before they can be executed. The result of preparation is the executable or operational form of the statement. The method of preparing an SQL statement and the persistence of its operational form distinguish static SQL from dynamic SQL.

Embedded SQL

Embedded SQL statements are SQL statements written within application programming languages such as C and preprocessed by an SQL preprocessor before the application program is compiled. There are two types of embedded SQL: static and dynamic.

Static SQL

The source form of a static SQL statement is embedded within an application program written in a host language such as COBOL. The statement is prepared before the program is executed and the operational form of the statement persists beyond the execution of the program.

A source program containing static SQL statements must be processed by an SQL precompiler before it is compiled. The precompiler turns the SQL statements into host language comments, and generates host language statements to invoke the database manager. The syntax of the SQL statements is checked during the precompile process.

The preparation of an SQL application program includes precompilation, the binding of its static SQL statements to the target database, and compilation of the modified source program. The steps are specified in the *Application Development Guide*.

Dynamic SQL

Programs containing embedded dynamic SQL statements must be precompiled like those containing static SQL, but unlike static SQL, the dynamic SQL statements are constructed and prepared at run time. The SQL statement text is prepared and executed using either the PREPARE and EXECUTE statements, or the EXECUTE IMMEDIATE statement. The statement can also be executed with the cursor operations if it is a SELECT statement.

DB2 Call Level Interface (CLI) & Open Database Connectivity (ODBC)

The DB2 Call Level Interface is an application programming interface in which functions are provided to application programs to process dynamic SQL statements. CLI programs can also be compiled using an Open Database Connectivity (ODBC) Software Developer’s Kit, available from Microsoft or other vendors, enabling access to ODBC data sources. Unlike using embedded SQL, no precompilation is required. Applications developed using this interface may be executed on a variety of databases without being compiled against each of the databases. Through the interface, applications use procedure calls at execution time to connect to databases, to issue SQL statements, and to get returned data and status information.

The DB2 CLI interface provides many features not available in embedded SQL. A few of these are:

- CLI provides function calls which support a consistent way to query and retrieve database system catalog information across the DB2 family of database management systems. This reduces the need to write database server specific catalog queries.
- CLI provides the ability to scroll through a cursor:
  - Forward by one or more rows
  - Backward by one or more rows
  - Forward from the first row by one or more rows
  - Backward from the last row by one or more rows
  - From a previously stored location in the cursor
- Stored procedures called from application programs written using CLI can return result sets to those programs.

The *CLI Guide and Reference* describes the APIs supported with this interface.

Java Database Connectivity (JDBC) and Embedded SQL for Java (SQLJ) Programs

DB2 Universal Database implements two standards-based Java programming APIs: Java Database Connectivity (JDBC) and embedded SQL for Java (SQLJ). Both can be used to create Java applications and applets that access DB2.

JDBC calls are translated to calls to DB2 CLI through Java native methods. JDBC requests flow from the DB2 client through DB2 CLI to the DB2 server. Static SQL cannot be used by JDBC.

SQLJ applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can also contain embedded static SQL statements in the SQLJ source files. An SQLJ source file has to be translated with the SQLJ translator before the resulting Java source code can be compiled.

For more information on JDBC and SQLJ applications, refer to the *Application Development Guide*.

Interactive SQL

Interactive SQL statements are entered by a user through an interface like the command line processor or the command center. These statements are processed as dynamic SQL statements. For example, an interactive SELECT statement can be processed dynamically using the DECLARE CURSOR, PREPARE, DESCRIBE, OPEN, FETCH, and CLOSE statements.

The *Command Reference* lists the commands that can be issued using the command line processor or similar facilities and products.

Schemas

A *schema* is a collection of named objects. Schemas provide a logical classification of objects in the database. Some of the objects that a schema may contain include tables, views, nicknames, triggers, functions and packages.

A schema is also an object in the database. It is explicitly created using the CREATE SCHEMA statement with a user recorded as owner. It can also be implicitly created when another object is created, provided the user has IMPLICIT_SCHEMA authority.

A *schema name* is used as the high-order part of a two-part object name. An object that is contained in a schema is assigned to the schema when the object is created. The schema to which it is assigned is determined by the name of the object if specifically qualified with a schema name or by the default schema name if not qualified.

For example, a user with DBADM authority creates a schema called C for user A.

```
CREATE SCHEMA C AUTHORIZATION A
```

User A can then issue the following statement to create a table called X in schema C:

```
CREATE TABLE C.X (COL1 INT)
```

Controlling Use of Schemas

When a database is created, all users have IMPLICIT_SCHEMA authority. This allows any user to create objects in any schema that does not already exist. An implicitly created schema allows any user to create other objects in this schema.[1]

1. The default privileges on an implicitly created schema provide upward compatibility with previous versions. Alias, distinct type, function and trigger creation is extended to implicitly created schemas.

If IMPLICIT_SCHEMA authority is revoked from PUBLIC, schemas are either explicitly created using the CREATE SCHEMA statement or implicitly created by users (such as those with DBADM authority) who are granted IMPLICIT_SCHEMA authority. While revoking IMPLICIT_SCHEMA authority from PUBLIC increases control over the use of schema names, it may result in authorization errors in existing applications when they attempt to create objects.

There are also privileges associated with a schema that control which users have the privilege to create, alter and drop objects in the schema. A schema owner is initially given all of these privileges on a schema with the ability to grant them to others. An implicitly created schema is owned by the system and all users are initially given the privilege to create objects in such a schema. A user with DBADM or SYSADM authority can change the privileges held by users on any schema. Therefore, access to create, alter and drop objects in any schema (even one that is implicitly created) can be controlled.

Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows. The rows are not necessarily ordered within a table (order is determined by the application program). At the intersection of every column and row is a specific data item called a *value*. A *column* is a set of values of the same type or one of its subtypes. A *row* is a sequence of values such that the *n*th value is a value of the *n*th column of the table.

A *base table* is created with the CREATE TABLE statement and is used to hold persistent user data. A *result table* is a set of rows that the database manager selects or generates from one or more base tables to satisfy a query.

A summary table is a table that is defined by a query that is also used to determine the data in the table. Summary tables can be used to improve the performance of queries. If the database manager determines that a portion of a query could be resolved using a summary table, the query may be rewritten by the database manager to use the summary table. This decision is based on certain settings such as the CURRENT REFRESH AGE and CURRENT QUERY OPTIMIZATION special registers.

A table can have the data type of each column defined separately, or have the types for the columns based on the attributes of a user-defined structured type. This is called a *typed table*. A user-defined structured type may be part of a type hierarchy. A *subtype* is said to inherit attributes from its *supertype*. Similarly, a typed table can be part of a table hierarchy. A *subtable* is said to inherit columns from its *supertable*. Note that the term *subtype* applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. A *proper subtype* of a structured type T is a structured type below T in the type hierarchy. Similarly the term *subtable* applies to a typed table and all typed tables that are below it in the table hierarchy. A *proper subtable* of a table T is a table below T in the table hierarchy.

A *declared temporary table* is created with a DECLARE GLOBAL TEMPORARY TABLE statement and is used to hold temporary data on behalf of a single application. This table is dropped implicitly when the application disconnects from the database.

Views

A *view* provides an alternative way of looking at the data in one or more tables.

A view is a named specification of a result table. The specification is a SELECT statement that is executed whenever the view is referenced in an SQL statement. Thus, a view can be thought of as having columns and rows just like a base table. For retrieval, all views can be used just like base tables. Whether a view can be used in an insert, update, or delete operation depends on its definition as explained in the description of CREATE VIEW. (See "CREATE VIEW" on page 823 for more information.)

When the column of a view is directly derived from a column of a base table, that column inherits any constraints that apply to the column of the base table. For example, if a view includes a foreign key of its base table, INSERT and UPDATE operations using that view are subject to the same referential constraint as the base table. Also, if the base table of a view is a parent table, DELETE and UPDATE operations using that view are subject to the same rules as DELETE and UPDATE operations on the base table.

A view can have the data type of each column derived from the result table, or have the types for the columns based on the attributes of a user-defined structured type. This is called a *typed view*. Similar to a typed table, a typed view can be part of a view hierarchy. A *subview* is said to inherit columns from its *superview*. The term *subview* applies to a typed view and all typed views that are below it in the view hierarchy. A *proper subview* of a view V is a view below V in the typed view hierarchy.

A view may become inoperative, in which case it is no longer available for SQL statements.

Aliases

An *alias* is an alternate name for a table or view. It can be used to reference a table or view in those cases where an existing table or view can be referenced.[2] Like tables and views, an alias may be created, dropped, and have comments associated with it. Aliases can also be created for nicknames. Unlike tables, aliases may refer to each other in a process called chaining. Aliases are publicly referenced names so no special authority or privilege is required to use an alias. Access to the tables and views referred to by the alias, however, still require the appropriate authorization for the current context.

In addition to table aliases, there are other types of aliases such as database and network aliases.

Refer to “Aliases” on page 71 and “CREATE ALIAS” on page 566 for more information about aliases.

Indexes

An *index* is an ordered set of pointers to rows of a base table. Each index is based on the values of data in one or more table columns. An index is an object that is separate from the data in the table. When an index is created, the database manager builds this structure and maintains it automatically.

Indexes are used by the database manager to:

- Improve performance. In most cases, access to data is faster than without an index.

  An index cannot be created for a view. However, an index created for a table on which a view is based may improve the performance of operations on the view.
- Ensure uniqueness. A table with a unique index cannot have rows with identical keys.

Keys

A *key* is a set of columns that can be used to identify or access a particular row or rows. The key is identified in the description of a table, index, or referential constraint. The same column can be part of more than one key.

A key composed of more than one column is called a *composite key*. In a table with a composite key, the ordering of the columns within the composite key is

2. An alias cannot be used in all contexts. For example, it cannot be used in the check condition of a check constraint. Also, an alias cannot reference a declared temporary table.

not constrained by their ordering within the table. The term *value* when used with respect to a composite key denotes a composite value. Thus, a rule such as "the value of the foreign key must be equal to the value of the primary key" means that each component of the value of the foreign key must be equal to the corresponding component of the value of the primary key.

Unique Keys

A *unique key* is a key that is constrained so that no two of its values are equal. The columns of a unique key cannot contain null values. The constraint is enforced by the database manager during the execution of any operation that changes data values, such as INSERT or UPDATE. The mechanism used to enforce the constraint is called a *unique index*. Thus, every unique key is a key of a unique index. Such an index is also said to have the UNIQUE attribute. See "Unique Constraints" on page 17 for a more detailed description.

Primary Keys

A *primary key* is a special case of a unique key. A table cannot have more than one primary key. See "Unique Keys" for a more detailed description.

Foreign Keys

A *foreign key* is a key that is specified in the definition of a referential constraint. See "Referential Constraints" on page 17 for a more detailed description.

Partitioning Keys

A *partitioning key* is a key that is part of the definition of a table in a partitioned database. The partitioning key is used to determine the partition on which the row of data is stored. If a partitioning key is defined, unique keys and primary keys must include the same columns as the partitioning key (they may have more columns). A table cannot have more than one partitioning key.

Constraints

A *constraint* is a rule that the database manager enforces.

There are three types of constraints:

- A *unique constraint* is a rule that forbids duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, a unique constraint could be defined on the supplier identifier in the supplier table to ensure that the same supplier identifier is not given to two suppliers.
- A *referential constraint* is a logical rule about values in one or more columns in one or more tables. For example, a set of tables shares information about a corporation's suppliers. Occasionally, a supplier's name changes. A referential constraint could be defined stating that the ID of the supplier in

a table must match a supplier id in the supplier information. This constraint prevents inserts, updates or deletes that would otherwise result in missing supplier information.

- A *table check constraint* sets restrictions on data added to a specific table. For example, it could define the salary level for an employee to never be less than $20,000.00 when salary data is added or updated in a table containing personnel information.

Referential and table check constraints may be turned on or off. Loading large amounts of data into the database is typically a time to turn off checking the enforcement of a constraint. The details of setting constraints on or off are discussed in "SET INTEGRITY" on page 1019.

Unique Constraints

A *unique constraint* is the rule that the values of a key are valid only if they are unique within the table. Unique constraints are optional and can be defined in the CREATE TABLE or ALTER TABLE statement using the PRIMARY KEY clause or the UNIQUE clause. The columns specified in a unique constraint must be defined as NOT NULL. A unique index is used by the database manager to enforce the uniqueness of the key during changes to the columns of the unique constraint.

A table can have an arbitrary number of unique constraints, with at most one unique constraint defined as a primary key. A table cannot have more than one unique constraint on the same set of columns.

A unique constraint that is referenced by the foreign key of a referential constraint is called the *parent key*.

When a unique constraint is defined in a CREATE TABLE statement, a unique index is automatically created by the database manager and designated as a primary or unique system-required index.

When a unique constraint is defined in an ALTER TABLE statement and an index exists on the same columns, that index is designated as unique and system-required. If such an index does not exist, the unique index is automatically created by the database manager and designated as a primary or unique system-required index.

Note that there is a distinction between defining a unique constraint and creating a unique index. Although both enforce uniqueness, a unique index allows nullable columns and generally cannot be used as a parent key.

Referential Constraints

Referential integrity is the state of a database in which all values of all foreign keys are valid. A *foreign key* is a column or set of columns in a table whose

values are required to match at least one primary key or unique key value of a row of its parent table. A *referential constraint* is the rule that the values of the foreign key are valid only if:
- they appear as values of a parent key, or
- some component of the foreign key is null.

The table containing the parent key is called the *parent table* of the referential constraint, and the table containing the foreign key is said to be a *dependent* of that table.

Referential constraints are optional and can be defined in CREATE TABLE statements and ALTER TABLE statements. Referential constraints are enforced by the database manager during the execution of INSERT, UPDATE, DELETE, ALTER TABLE ADD CONSTRAINT, and SET INTEGRITY statements. The enforcement is effectively performed at the completion of the statement.

Referential constraints with a delete or update rule of RESTRICT are enforced before all other referential constraints. Referential constraints with a delete or update rule of NO ACTION behave like RESTRICT in most cases. However, in certain SQL statements there can be a difference.

Note that referential integrity, check constraints and triggers can be combined in execution. For further information on the combination of these elements, see "Appendix J. Interaction of Triggers and Constraints" on page 1287.

The rules of referential integrity involve the following concepts and terminology:

Parent key	A primary key or unique key of a referential constraint.
Parent row	A row that has at least one dependent row.
Parent table	A table that contains the parent key of a referential constraint. A table can be a parent in an arbitrary number of referential constraints. A table which is the parent in a referential constraint may also the dependent of a referential constraint.
Dependent table	A table that contains at least one referential constraint in its definition. A table can be a dependent in an arbitrary number of referential constraints. A table which is the dependent in a referential constraint may also the parent of a referential constraint.

Descendent table	A table is a descendent of table T if it is a dependent of T or a descendent of a dependent of T.
Dependent row	A row that has at least one parent row.
Descendent row	A row is a descendent of row p if it is a dependent of p or a descendent of a dependent of p.
Referential cycle	A set of referential constraints such that each table in the set is a descendent of itself.
Self-referencing row	A row that is a parent of itself.
Self-referencing table	A table that is a parent and a dependent in the same referential constraint. The constraint is called a *self-referencing constraint*.

Insert Rule

The insert rule of a referential constraint is that a non-null insert value of the foreign key must match some value of the parent key of the parent table. The value of a composite foreign key is null if any component of the value is null. This rule is implicit when a foreign key is specified.

Update Rule

The update rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION and RESTRICT. The update rule applies when a row of the parent or a row of the dependent table is updated.

In the case of a parent row, when a value in a column of the parent key is updated:

- if any row in the dependent table matched the original value of the key, the update is rejected when the update rule is RESTRICT
- if any row in the dependent table does not have a corresponding parent key when the update statement is completed (excluding AFTER triggers), the update is rejected when the update rule is NO ACTION.

In the case of a dependent row, the update rule that is implicit when a foreign key is specified is NO ACTION. NO ACTION means that a non-null update value of a foreign key must match some value of the parent key of the parent table when the update statement is completed.

The value of a composite foreign key is null if any component of the value is null.

Delete Rule

The delete rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION, RESTRICT, CASCADE, or SET NULL. SET NULL can be specified only if some column of the foreign key allows null values.

The delete rule of a referential constraint applies when a row of the parent table is deleted. More precisely, the rule applies when a row of the parent table is the object of a delete or propagated delete operation (defined below) and that row has dependents in the dependent table of the referential constraint. Let P denote the parent table, let D denote the dependent table, and let p denote a parent row that is the object of a delete or propagated delete operation. If the delete rule is:

- RESTRICT or NO ACTION, an error occurs and no rows are deleted
- CASCADE, the delete operation is propagated to the dependents of p in D
- SET NULL, each nullable column of the foreign key of each dependent of p in D is set to null

Each referential constraint in which a table is a parent has its own delete rule, and all applicable delete rules are used to determine the result of a delete operation. Thus, a row cannot be deleted if it has dependents in a referential constraint with a delete rule of RESTRICT or NO ACTION or the deletion cascades to any of its descendents that are dependents in a referential constraint with the delete rule of RESTRICT or NO ACTION.

The deletion of a row from parent table P involves other tables and may affect rows of these tables:

- If table D is a dependent of P and the delete rule is RESTRICT or NO ACTION, D is involved in the operation but is not affected by the operation.
- If D is a dependent of P and the delete rule is SET NULL, D is involved in the operation, and rows of D may be updated during the operation.
- If D is a dependent of P and the delete rule is CASCADE, D is involved in the operation and rows of D may be deleted during the operation.

  If rows of D are deleted, the delete operation on P is said to be propagated to D. If D is also a parent table, the actions described in this list apply, in turn, to the dependents of D.

Any table that may be involved in a delete operation on P is said to be *delete-connected* to P. Thus, a table is delete-connected to table P if it is a dependent of P, or a dependent of a table to which delete operations from P cascade.

Table Check Constraints

A *table check constraint* is a rule that specifies the values allowed in one or more columns of every row of a table. They are optional and can be defined using the SQL statements CREATE TABLE and ALTER TABLE. The specification of table check constraints is a restricted form of a search condition. One of the restrictions is that a column name in a table check constraint on table *T* must identify a column of *T*.

A table can have an arbitrary number of table check constraints. They are enforced when:
- a row is inserted into the table
- a row of the table is updated.

A table check constraint is enforced by applying its search condition to each row that is inserted or updated. An error occurs if the result of the search condition is false for any row.

When one or more table check constraints are defined in the ALTER TABLE statement for a table with existing data, the existing data is checked against the new condition before the ALTER TABLE statement succeeds. The table can be placed in *check pending state* which will allow the ALTER TABLE statement to succeed without checking the data. The SET INTEGRITY statement is used to place the table into check pending state. It is also used to resume the checking of each row against the constraint.

Triggers

A *trigger* defines a set of actions that are executed at, or triggered by, a delete, insert, or update operation on a specified table. When such an SQL operation is executed, the trigger is said to be *activated*.

Triggers can be used along with referential constraints and check constraints to enforce data integrity rules. Triggers can also be used to cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts.

Triggers are a useful mechanism to define and enforce *transitional* business rules which are rules that involve different states of the data (for example, salary cannot be increased by more than 10 percent). For rules that do not involve more than one state of the data, check and referential integrity constraints should be considered.

Using triggers places the logic to enforce the business rules in the database and relieves the applications using the tables from having to enforce it.

Centralized logic enforced on all the tables means easier maintenance, since no application program changes are required when the logic changes.

Triggers are optional and are defined using the CREATE TRIGGER statement.

There are a number of criteria that are defined when creating a trigger which are used to determine when a trigger should be activated.

- The *subject table* defines the table for which the trigger is defined.
- The *trigger event* defines a specific SQL operation that modifies the subject table. The operation could be delete, insert or update.
- The *trigger activation time* defines whether the trigger should be activated before or after the trigger event is performed on the subject table.

The statement that causes a trigger to be activated will include a *set of affected rows*. These are the rows of the subject table that are being deleted, inserted or updated. The *trigger granularity* defines whether the actions of the trigger will be performed once for the statement or once for each of the rows in the set of affected rows.

The *triggered action* consists of an optional search condition and a set of SQL statements that are executed whenever the trigger is activated. The SQL statements are only executed if the search condition evaluates to true. When the trigger activation time is before the trigger event, triggered actions can include statements that select, set transition variables, and signal SQLSTATEs. When the trigger activation time is after the trigger event, triggered actions can include statements that select, update, insert, delete, and signal SQLSTATEs.

The triggered action may refer to the values in the set of affected rows. This is supported through the use of *transition variables*. Transition variables use the names of the columns in the subject table qualified by a specified name that identifies whether the reference is to the old value (prior to the update) or the new value (after the update). The new value can also be changed using the SET transition-variable statement in before update or insert triggers. Another means of referring to the values in the set of affected rows is using *transition tables*. Transition tables also use the names of the columns of the subject table but have a name specified that allows the complete set of affected rows to be treated as a table. Transition tables can only be used in after triggers; and separate transition tables can be defined for old and new values.

Multiple triggers can be specified for a combination of table, event, or activation time. The order in which the triggers are activated is the same as the order in which they were created. Thus, the most recently created trigger will be the last trigger activated.

The activation of a trigger may cause *trigger cascading*. This is the result of the activation of one trigger that executes SQL statements that cause the activation of other triggers or even the same trigger again. The triggered actions may also cause updates as a result of the original modification, or as a result of referential integrity delete rules which may result in the activation of additional triggers. With trigger cascading, a significant chain of triggers and referential integrity delete rules may be activated causing significant change to the database as a result of a single delete, insert or update statement.

Event Monitors

An *event monitor* tracks specific data as the result of an event. For example, starting the database might be an event that causes an event monitor to track the number of users on the system by taking an hourly snapshot of authorization IDs using the database.

Event monitors are activated or deactivated by a statement (SET EVENT MONITOR STATE). A function (EVENT_MON_STATE) can be used to find the current state of an event monitor; that is, if it is active or not active.

Queries

A *query* is a component of certain SQL statements that specifies a (temporary) result table.

Table Expressions

A *table expression* creates a (temporary) result table from a simple query. Clauses further refine the result table. For example, a table expression could be a query that selects all the managers from several departments and further specifies that they have over 15 years of working experience and are located at the New York branch office.

Common Table Expressions

A *common table expression* is like a temporary view within a complex query, and can be referenced in other places within the query; for example, in place of a view, to avoid creating the view. Each use of a specific common table expression within a complex query shares the same temporary view.

Recursive use of a common table expression within a query can be used to support applications such as bill of materials (BOM), airline reservation systems, and network planning. A set of examples from a BOM application is contained in “Appendix M. Recursion Example: Bill of Materials” on page 1329.

Packages

A *package* is an object that contains all the *sections* from a single source file. A section is the compiled form of an SQL statement. While every section corresponds to one statement, every statement does not necessarily have a section. The sections created for static SQL can be thought of as the bound or operational form of SQL statements. The sections created for dynamic SQL can be thought of as placeholder control structures which are used at execution time. Packages are produced during program preparation. See the *Application Development Guide* for more information on packages.

Catalog Views

The database manager maintains a set of views and base tables that contain information about the data under its control. These views and base tables are collectively known as the *catalog*. They contain information about objects in the database such as tables, views, indexes, packages and functions.

The catalog views are like any other database views. SQL statements can be used to look at the data in the catalog views in the same way that data is retrieved from any other view in the system. The database manager ensures that the catalog contains accurate descriptions of the objects in the database at all times. A set of updatable catalog views can be used to modify certain values in the catalog (see “Updatable Catalog Views” on page 1128).

Statistical information is also contained in the catalog. The statistical information is updated by utilities executed by an administrator, or through update statements by appropriately authorized users.

The catalog views are listed in “Appendix D. Catalog Views” on page 1127.

Application Processes, Concurrency, and Recovery

All SQL programs execute as part of an *application process* or agent. An application process involves the execution of one or more programs, and is the unit to which the database manager allocates resources and locks. Different application processes may involve the execution of different programs, or different executions of the same program.

More than one application process may request access to the same data at the same time. *Locking* is the mechanism used to maintain data integrity under such conditions, preventing, for example, two application processes from updating the same row of data simultaneously.

The database manager acquires locks in order to prevent uncommitted changes made by one application process from being accidentally perceived by any other. The database manager releases all locks it has acquired and retained on behalf of an application process when that process ends. However, an application process can explicitly request that locks be released sooner. This is done using a *commit* operation which releases locks acquired during the unit of work and also commits database changes made during the unit of work.

The database manager provides a means of *backing out* uncommitted changes made by an application process. This might be necessary in the event of a failure on the part of an application process, or in a *deadlock* or lock time-out situation. An application process itself, however, can explicitly request that its database changes be backed out. This operation is called *rollback*.

A *unit of work* is a recoverable sequence of operations within an application process. A unit of work is initiated when an application process is started[3]. A unit of work is also initiated when the previous unit of work is ended by something other than the termination of the application process. A unit of work is ended by a commit operation, a rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes made within the unit of work it ends.

While these changes remain uncommitted, other application processes are unable to perceive them and they can be backed out.[4] Once committed, these database changes are accessible by other application processes and can no longer be backed out by a rollback.

Locks acquired by the database manager on behalf of an application process are held until the end of a unit of work. The exceptions to this rule are cursor stability isolation level, where the lock is released as the cursor moves from row to row, and uncommitted read level, where locks are not obtained (see "Isolation Level" on page 27).

The initiation and termination of a unit of work define points of consistency within an application process. For example, a banking transaction might involve the transfer of funds from one account to another.

---

3. DB2 CLI and embedded SQL support a connection mode called *concurrent transactions* which supports multiple connections, each of which is an independent transaction. An application can have multiple concurrent connections to the same database. Refer to the *Application Development Guide* for details on multiple thread database access.

4. Except for isolation level uncommitted read, described in "Uncommitted Read (UR)" on page 29.

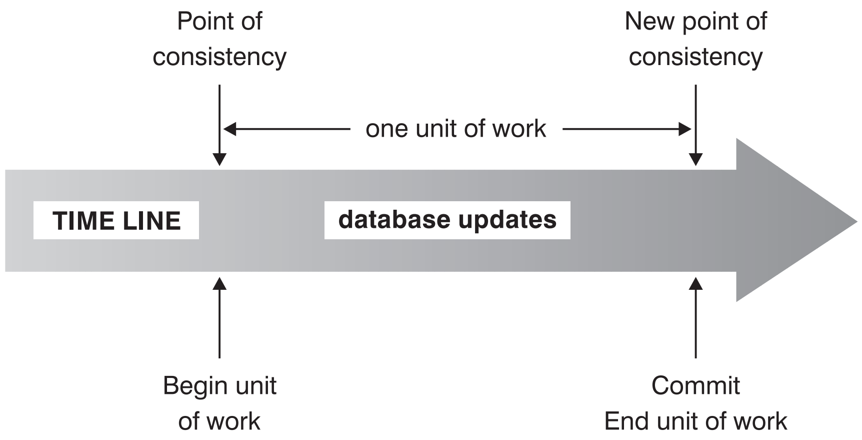


Figure 1. Unit of Work with a Commit Statement

Such a transaction would require that these funds be subtracted from the first account, and added to the second.
Following the subtraction step, the data is inconsistent. Only after the funds

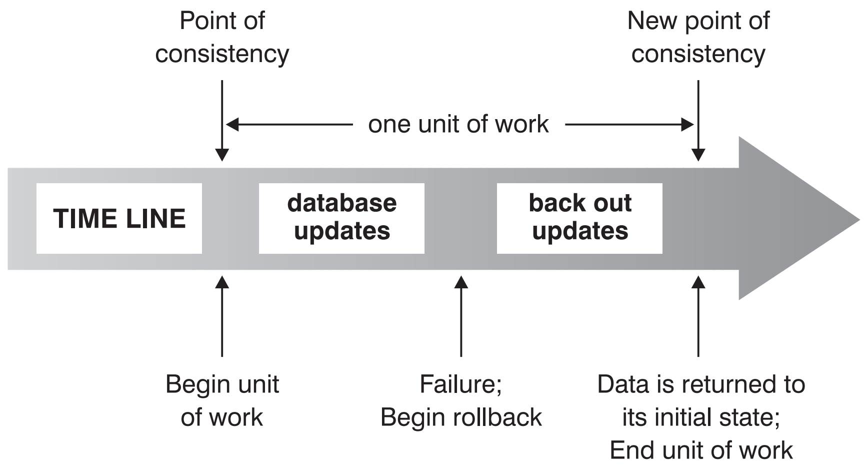


Figure 2. Unit of Work with a Rollback Statement

have been added to the second account is consistency reestablished. When both steps are complete, the commit operation can be used to end the unit of work, thereby making the changes available to other application processes.

If a failure occurs before the unit of work ends, the database manager will roll back uncommitted changes to restore the data consistency that it assumes existed when the unit of work was initiated.

Note: An application process is never prevented from performing operations because of its own locks.[5]

Isolation Level

The *isolation level* associated with an application process defines the degree of isolation of that application process from other concurrently executing application processes. The isolation level of an application process, P, therefore specifies:

- The degree to which rows read and updated by P are available to other concurrently executing application processes
- The degree to which update activity of other concurrently executing application processes can affect P.

The isolation level is specified as an attribute of a package and applies to the application processes that use the package. The isolation level is specified in the program preparation process. Depending on the type of lock, this limits or prevents access to the data by concurrent application processes. For details on different types and attributes of specific locks refer to the *Administration Guide*. Declared temporary tables and the rows of declared temporary tables are not locked at all because they are only accessible by the application that declared the temporary tables. Thus, the following discussion on locking and isolation levels does not apply to declared temporary tables.

The database manager supports three general categories of locks:

Share	Limits concurrent application processes to read-only operations on the data.
Update	Limits concurrent application processes to read-only operations on the data providing these processes have not declared they might update the row. The database manager assumes the process looking at the row presently may update the row.
Exclusive	Prevents concurrent application processes from accessing the data in any way except for application processes with an isolation level of *uncommitted read*, which can read but not modify the data. (See “Uncommitted Read (UR)” on page 29.)

5. If an application is using concurrent transactions, then the locks from one transaction may affect the operation of a concurrent transaction. See the *Application Development Guide* for details.

Locking occurs at the base table row. The database manager, however, can replace multiple row locks with a single table lock. This is called *lock escalation*. An application process is guaranteed at least the minimum requested lock level.

The DB2 Universal Database database manager supports four isolation levels. Regardless of the isolation level, the database manager places exclusive locks on every row that is inserted, updated, or deleted. Thus, all isolation levels ensure that any row that is changed by this application process during a unit of work is not changed by any other application processes until the unit of work is complete. The isolation levels are:

Repeatable Read (RR)

Level RR ensures that:

- Any row read during a unit of work[6] is not changed by other application processes until the unit of work is complete.[7]
- Any row changed by another application process cannot be read until it is committed by that application process.

RR does not allow phantom rows (see Read Stability) to be seen.

In addition to any exclusive locks, an application process running at level RR acquires at least share locks on all the rows it references. Furthermore, the locking is performed so that the application process is completely isolated from the effects of concurrent application processes.

Read Stability (RS)

Like level RR, level RS ensures that:

- Any row read during a unit of work[6] is not changed by other application processes until the unit of work is complete[8]
- Any row changed by another application process cannot be read until it is committed by that application process.

Unlike RR, RS does not completely isolate the application process from the effects of concurrent application processes. At level RS, application processes that issue the same query more than once might see additional rows. These additional rows are called *phantom rows*.

6. The rows must be read in the same unit of work as the corresponding OPEN statement. See WITH HOLD in "DECLARE CURSOR" on page 841.

7. Use of the optional WITH RELEASE clause on the CLOSE statement means that any guarantees against non-repeatable read and phantoms no longer apply to any previously accessed rows if the cursor is reopened.

8. Use of the optional WITH RELEASE clause on the CLOSE statement means that any guarantees against non-repeatable read no longer apply to any previously accessed rows if the cursor is reopened.

For example, a phantom row can occur in the following situation:

1. Application process P1 reads the set of rows *n* that satisfy some search condition.
2. Application process P2 then INSERTs one or more rows that satisfy the search condition and COMMITs those INSERTs.
3. P1 reads the set of rows again with the same search condition and obtains both the original rows and the rows inserted by P2.

In addition to any exclusive locks, an application process running at level RS acquires at least share locks on all the qualifying rows.

Cursor Stability (CS)

Like the RR level:

- CS ensures that any row that was changed by another application process cannot be read until it is committed by that application process.

Unlike the RR level:

- CS only ensures that the current row of every updatable cursor is not changed by other application processes. Thus, the rows that were read during a unit of work can be changed by other application processes.

In addition to any exclusive locks, an application process running at level CS has at least a share lock for the current row of every cursor.

Uncommitted Read (UR)

For a SELECT INTO, FETCH with a read-only cursor, fullselect used in an INSERT, row fullselect in an UPDATE, or scalar fullselect (wherever used), level UR allows:

- Any row that is read during the unit of work to be changed by other application processes.
- Any row that was changed by another application process to be read even if the change has not been committed by that application process.

For other operations, the rules of level CS apply.

Comparison of Isolation Levels

A comparison of the four isolation levels can be found on "Appendix I. Comparison of Isolation Levels" on page 1285.

Distributed Relational Database

A *distributed relational database* consists of a set of tables and other objects that are spread across different but interconnected computer systems. Each computer system 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 to execute SQL statements on another computer system.

Distributed relational databases are built on formal requester-server protocols and functions. An *application requester* supports the application end of a connection. It transforms a database request from the application into communication protocols suitable for use in the distributed database network. These requests are received and processed by an *application server* at the other end of the connection. Working together, the application requester and application server handle the communication and location considerations so that the application is isolated from these considerations and can operate as if it were accessing a local database. A simple distributed relational database environment is illustrated in Figure 3.

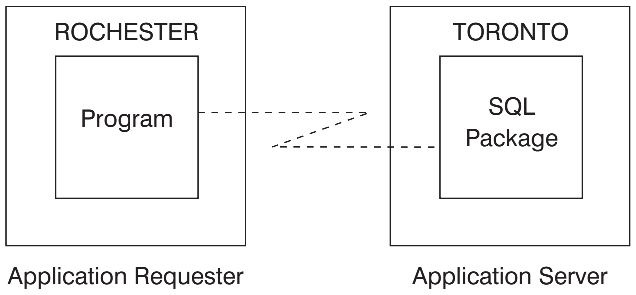


Figure 3. A Distributed Relational Database Environment

For more information on Distributed Relational Database Architecture (DRDA) communication protocols, see *Distributed Relational Database Architecture Reference* SC26-4651.

Application Servers

An application process must be connected to the application server of a database manager before SQL statements that reference tables or views can be executed. A CONNECT statement establishes a connection between an application process and its server.[9] The server can change when a CONNECT statement is executed.

The application server can be local to or remote from the environment where the process is initiated. (An application server is present, even when not using distributed relational databases.) This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement. For a description of local directories, see the *Administration Guide*

9. DB2 CLI and embedded SQL support a connection mode called *concurrent transactions* which supports multiple connections, each of which is an independent transaction. An application can have multiple concurrent connections to the same database. Refer to the *Application Development Guide* for details on multiple thread database access.

To execute a static SQL statement that references tables or views, the application server uses the bound form of the statement. This bound statement is taken from a package that the database manager previously created through a bind operation.

For the most part, an application can use the statements and clauses that are supported by the database manager of the application server to which it is currently connected, even though that application might be running via the application requester of a database manager that does not support some of those statements and clauses.

For information about using an application server to submit queries in a system of distributed data sources, see "Server Definitions and Server Options" on page 44.

CONNECT (Type 1) and CONNECT (Type 2)

There are two types of CONNECT statements:

- CONNECT (Type 1) supports the single database per unit of work (Remote Unit of Work) semantics. See "CONNECT (Type 1)" on page 550.
- CONNECT (Type 2) supports the multiple databases per unit of work (Application-Directed Distributed Unit of Work) semantics. See "CONNECT (Type 2)" on page 558.

Remote Unit of Work

The *remote unit of work* facility provides for the remote preparation and execution of SQL statements. An application process at computer system A can connect to an application server at computer system B and, within one or more units of work, execute any number of static or dynamic SQL statements that reference objects at B. After ending a unit of work at B, the application process can connect to an application server at computer system C, and so on.

Most SQL statements can be remotely prepared and executed with the following restrictions:

- All objects referenced in a single SQL statement must be managed by the same application server
- All of the SQL statements in a unit of work must be executed by the same application server

Remote Unit of Work Connection Management

This section outlines the connection states that an application process may enter.

Connection States:

An application process is in one of four states at any time:

Connectable and connected

Unconnectable and connected

Connectable and unconnected
Implicitly connectable (if implicit connect is available).

If implicit connect is available (see Figure 4 on page 34), the application process is initially in the *implicitly connectable* state. If implicit connect is not available (see Figure 5 on page 35), the application process is initially in the *connectable and unconnected* state.

Availability of implicit connect is determined by installation options, environment variables, and authentication settings. See the *Quick Beginnings* for information on setting implicit connect on installation and the *Administration Guide* for information on environment variables and authentication settings.

The implicitly connectable state:
If implicit connect is available, this is the initial state of an application process. The CONNECT RESET statement causes a transition to this state. Issuing a COMMIT or ROLLBACK statement in the unconnectable and connected state followed by a DISCONNECT statement in the connectable and connected state also results in this state.

The connectable and connected state:
An application process is connected to an application server and CONNECT statements can be executed.

If implicit connect is available:

- The application process enters this state when a CONNECT TO statement or a CONNECT without operands statement is successfully executed from the connectable and unconnected state.
- The application process may also enter this state from the implicitly connectable state if any SQL statement other than CONNECT RESET, DISCONNECT, SET CONNECTION, or RELEASE is issued.

Whether or not implicit connect is available, this state is entered when:

- A CONNECT TO statement is successfully executed from the connectable and unconnected state.
- A COMMIT or ROLLBACK statement is successfully issued or a forced rollback occurs from the unconnectable and connected state.

The unconnectable and connected state:
An application process is connected to an application server, but a CONNECT TO statement cannot be successfully executed to change application servers. The process enters this state from the connectable and connected state when it executes any SQL statement other than the

following statements: CONNECT TO, CONNECT with no operand, CONNECT RESET, DISCONNECT, SET CONNECTION, RELEASE, COMMIT or ROLLBACK.

The connectable and unconnected state:

An application process is not connected to an application server. The only SQL statement that can be executed is CONNECT TO, otherwise an error (SQLSTATE 08003) is raised.

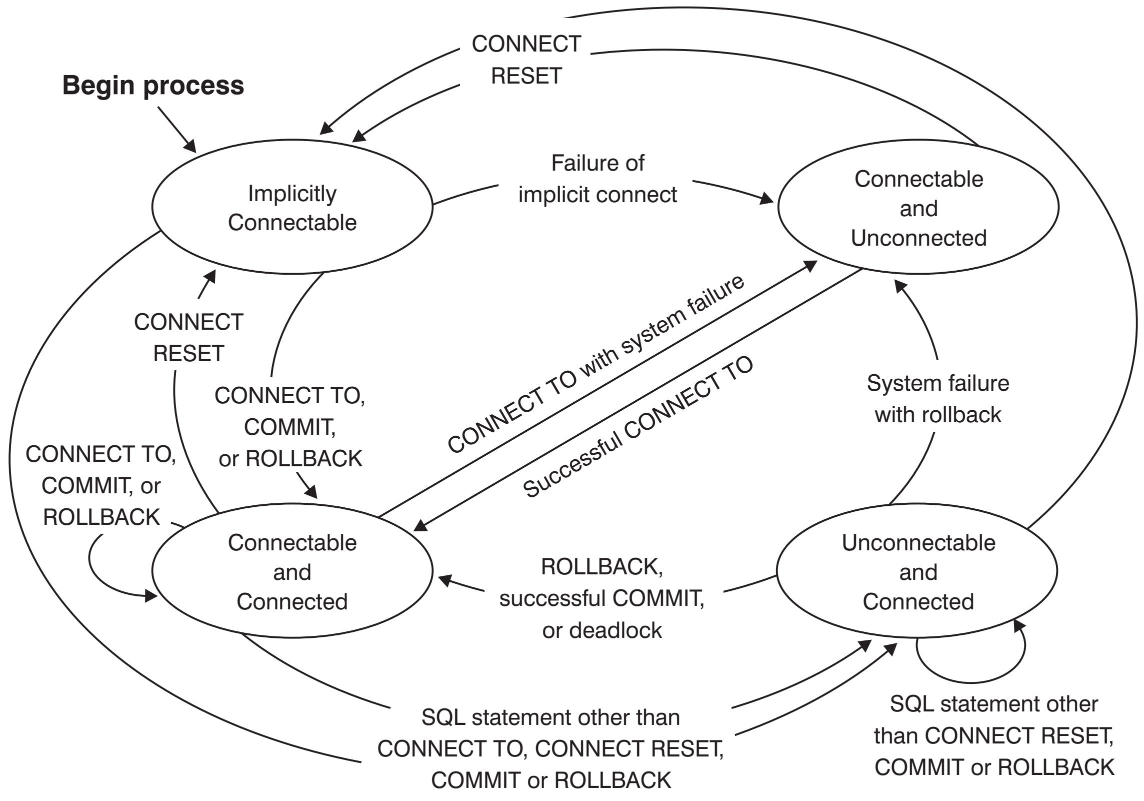
Whether or not implicit connect is available:

– The application process enters this state if an error occurs when a CONNECT TO statement is issued or an error occurs in a unit of work which causes the loss of a connection and a rollback. An error caused because the application process is not in the connectable state or the server-name is not listed in the local directory does not cause a transition to this state.

If implicit connect is not available:

– the CONNECT RESET and DISCONNECT statements cause a transition to this state.

State Transitions are shown in the following diagrams.



CONNECT RESET
Begin process
Implicitly Connectable
Failure of implicit connect
Connectable and Unconnected
CONNECT RESET
CONNECT TO with system failure
Successful CONNECT TO
CONNECT TO, COMMIT, or ROLLBACK
System failure with rollback
CONNECT TO, COMMIT, or ROLLBACK
Connectable and Connected
ROLLBACK, successful COMMIT, or deadlock
Unconnectable and Connected
SQL statement other than CONNECT TO, CONNECT RESET, COMMIT or ROLLBACK
SQL statement other than CONNECT RESET, COMMIT or ROLLBACK

Figure 4. Connection State Transitions If Implicit Connect Is Available

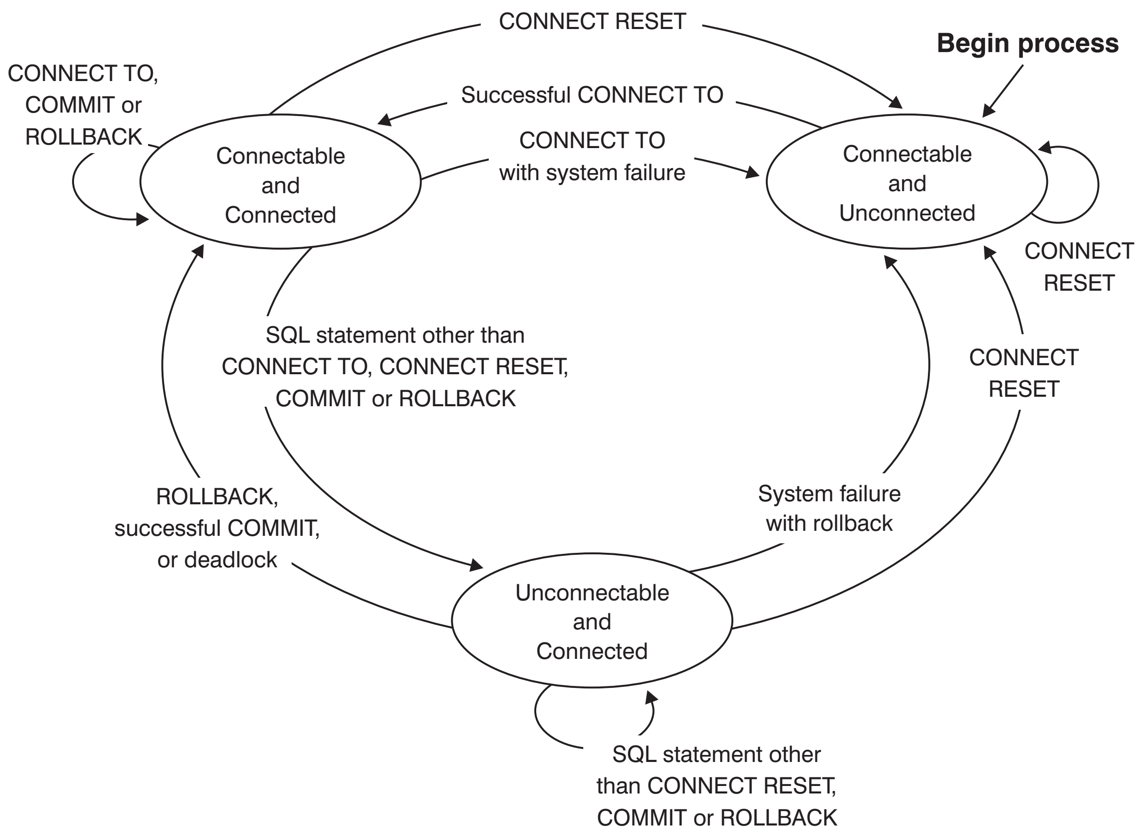

CONNECT RESET
Begin process
CONNECT TO, COMMIT or ROLLBACK
Successful CONNECT TO
Connectable and Connected
CONNECT TO with system failure
Connectable and Unconnected
CONNECT RESET
SQL statement other than CONNECT TO, CONNECT RESET, COMMIT or ROLLBACK
CONNECT RESET
ROLLBACK, successful COMMIT, or deadlock
System failure with rollback
Unconnectable and Connected
SQL statement other than CONNECT RESET, COMMIT or ROLLBACK

Figure 5. Connection State Transitions If Implicit Connect Is Not Available

Additional Rules:

- It is not an error to execute consecutive CONNECT statements because CONNECT itself does not remove the application process from the connectable state.
- It is an error to execute consecutive CONNECT RESET statements.
- It is an error to execute any SQL statement other than CONNECT TO, CONNECT RESET, CONNECT with no operand, SET CONNECTION, RELEASE, COMMIT, or ROLLBACK, and then execute a CONNECT TO statement. To avoid the error, a CONNECT RESET, DISCONNECT (preceded by a COMMIT or ROLLBACK statement), COMMIT, or ROLLBACK statement should be executed before executing the CONNECT TO.

Application-Directed Distributed Unit of Work

The *application-directed distributed unit of work facility* also provides for the remote preparation and execution of SQL statements in the same fashion as remote unit of work. An application process at computer system A can connect to an application server at computer system B by issuing a CONNECT or SET CONNECTION statement. The application process can then execute any number of static and dynamic SQL statements that reference objects at B before ending the unit of work. All objects referenced in a single SQL statement must be managed by the same application server. However, unlike remote unit of work, any number of application servers can participate in the same unit of work. A commit or rollback operation ends the unit of work.

Application-Directed Distributed Unit of Work Connection Management

An application-directed distributed unit of work uses a Type 2 connection. A Type 2 connection connects an application process to the identified application server and establishes the rules for application-directed distributed unit of work.

Overview of Application Process and Connection States

At any time a type 2 application process:

– Is always connectable

– Is in the *connected* state or *unconnected* state.

– Has a set of zero or more connections.

Each connection of an application process is uniquely identified by the database alias of the application server of the connection.

At any time an individual connection has one of the following sets of connection states:

– *current* and *held*
– *current* and *release-pending*
– *dormant* and *held*
– *dormant* and *release-pending*

Initial States and State Transitions: A type 2 application process is initially in the *unconnected state* and does not have any connections.

A connection initially is in the *current* and *held state*.

The following diagram shows the state transitions:

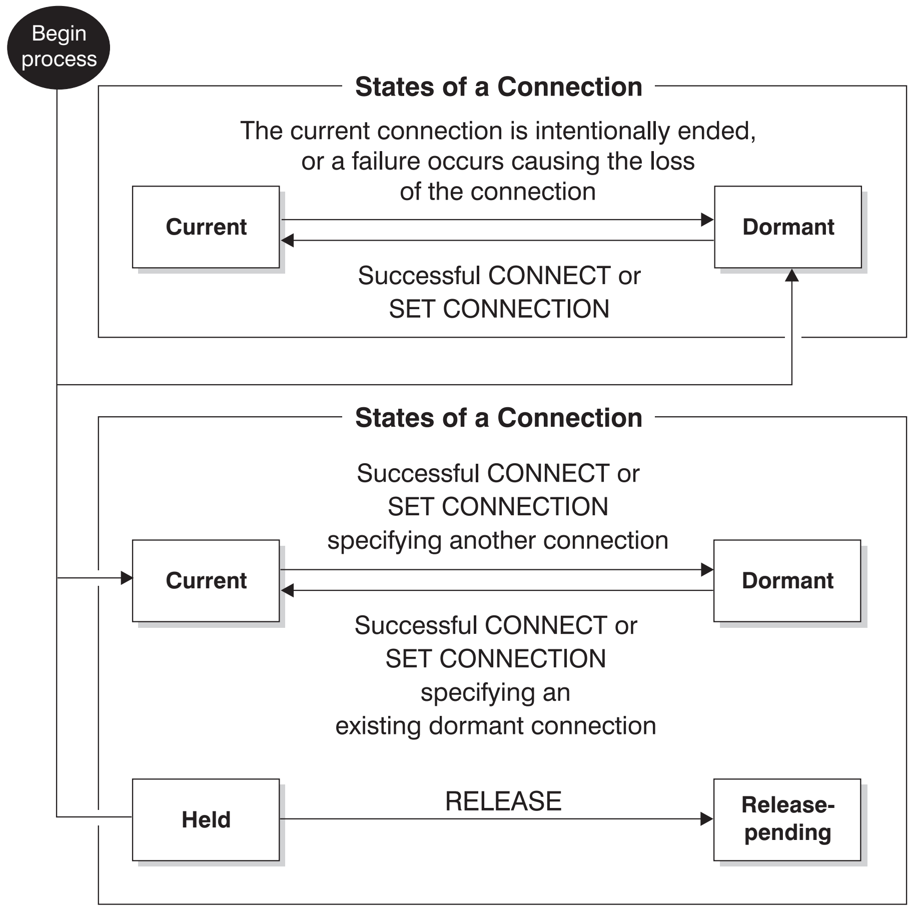

Begin process
States of a Connection
The current connection is intentionally ended, or a failure occurs causing the loss of the connection
Current
Dormant
Successful CONNECT or SET CONNECTION
States of a Connection
Successful CONNECT or SET CONNECTION specifying another connection
Current
Dormant
Successful CONNECT or SET CONNECTION specifying an existing dormant connection
Held
RELEASE
Release-pending

Figure 6. Application-Directed Distributed Unit of Work Connection and Application Process Connection State Transitions

Application Process Connection States: A different application server can be established by the explicit or implicit execution of a CONNECT statement. [10] The following rules apply:

- A context can not have more than one connection to the same application server at the same time. See *Administration Guide* and *Application Development Guide* for information on support of multiple connections to the same DB2 Universal Database at the same time.
- When an application process executes a SET CONNECTION statement, the specified location name must be an existing connection in the set of connections of the application process.

10. Note that a Type 2 implicit connection is more restrictive than a Type 1. See "CONNECT (Type 2)" on page 558 for details.

- When an application process executes a CONNECT statement, and the SQLRULES(STD) option is in effect the specified server name must not be an existing connection in the set of connections of the application process. See “Options that Govern Distributed Unit of Work Semantics” on page 39 for a description of the SQLRULES option.

If an application process has a current connection, the application process is in the *connected* state. The CURRENT SERVER special register contains the name of the application server of the current connection. The application process can execute SQL statements that refer to objects managed by that application server.

An application process in the unconnected state enters the connected state when it successfully executes a CONNECT or SET CONNECTION statement. If there is no connection in the application but SQL statements are issued, an implicit connect will be made provided the DB2DBDFT environment variable has been defined with a default database.

If an application process does not have a current connection, the application process is in the *unconnected* state. The only SQL statements that can be executed are CONNECT, DISCONNECT ALL, DISCONNECT specifying a database, SET CONNECTION, RELEASE, COMMIT and ROLLBACK.

An application process in the *connected state* enters the *unconnected state* when its current connection is intentionally ended or the execution of an SQL statement is unsuccessful because of a failure that causes a rollback operation at the application server and loss of the connection. Connections are intentionally ended either by the successful execution of a DISCONNECT statement or by the successful execution of a commit operation when the connection is in the *release-pending state*. Different options specified in the DISCONNECT precompiler option affect intentionally ending a connection. If set to AUTOMATIC, then all connections are ended. If set to CONDITIONAL, then all connections that do not have open WITH HOLD cursors are ended.

States of a Connection: If an application process executes a CONNECT statement and the server name is known to the application requester and is not in the set of existing connections of the application process, then:
- the current connection is placed into the *dormant state*, and
- the server name is added to the set of connections, and
- the new connection is placed into both the *current state* and the *held state*.

If the server name is already in the set of existing connections of the application process and the application is precompiled with the option SQLRULES(STD), an error (SQLSTATE 08002) is raised.

- ***Held and Release-pending States:*** The RELEASE statement controls whether a connection is in the held or release-pending state. A *release-pending state* means that a disconnect is to occur for the connection at the next successful commit operation (a rollback has no effect on connections). A *held state* means that a connection is not to be disconnected at the next operation. All connections are initially in the held state and may be moved into the release-pending state using the RELEASE statement. Once in the release-pending state, a connection cannot be moved back to the held state. A connection will remain in a release-pending state across unit of work boundaries if a ROLLBACK statement is issued or if an unsuccessful commit operation results in a rollback operation.

  Even if a connection is not explicitly marked for release, it may still be disconnected by a commit operation if the commit operation satisfies the conditions of the DISCONNECT precompiler option.
- ***Current and Dormant States:*** Regardless of whether a connection is in the *held state* or the *release-pending state*, a connection can also be in the *current state* or *dormant state*. A *current state* means that the connection is the one used for SQL statements that are executed while in this state. A *dormant state* means that the connection is not current. The only SQL statements which can flow on a dormant connection are COMMIT and ROLLBACK; or DISCONNECT and RELEASE, which can specify either ALL (for all connections) or a specific database name. The SET CONNECTION and CONNECT statements change the connection for the named server into the *current state* while any existing connections are either placed or remain in the *dormant state*. At any point in time, only one connection can be in the *current state*. When a dormant connection becomes current in the same unit of work, the state of all locks, cursors, and prepared statements will remain the same and reflect their last use when the connection was current.

When a Connection is Ended: When a connection is ended, all resources that were acquired by the application process through the connection and all resources that were used to create and maintain the connection are de-allocated. For example, if the application process executes a RELEASE statement, any open cursors will be closed when the connection is ended during the next commit operation.

A connection can also be ended because of a communications failure. The application process is placed in the unconnected state if the connection ended was the current one.

All connections of an application process are ended when the process ends.

Options that Govern Distributed Unit of Work Semantics

The semantics of type 2 connection management are determined by a set of precompiler options. These are summarized briefly below with the defaults indicated by bold and underlined text. For details refer to the *Command Reference* or *Administrative API Reference* manuals.

- CONNECT (**1** | 2)

  Specifies whether CONNECT statements are to be processed as type 1 or type 2.
- SQLRULES (**DB2** | STD)

  Specifies whether type 2 CONNECTs should be processed according to the DB2 rules which allow CONNECT to switch to a dormant connection, or the SQL92 Standard (STD) rules which do not allow this.
- DISCONNECT (**EXPLICIT** | CONDITIONAL | AUTOMATIC)

  Specifies what database connections are disconnected when a commit operation occurs. They are either:
  - those which had been explicitly marked for release by the SQL RELEASE statement (EXPLICIT), or
  - those that have no open WITH HOLD cursors as well as those marked for release (CONDITIONAL) [11] , or
  - all connections (AUTOMATIC).
- SYNCPOINT (**ONEPHASE** | TWOPHASE | NONE)

  Specifies how commits or rollbacks are to be coordinated among multiple database connections.

  ONEPHASE Updates can only occur on one database in the unit of work, all other databases are read-only. Any update attempts to other databases raise an error (SQLSTATE 25000).

  TWOPHASE A Transaction Manager (TM) will be used at run time to coordinate two phase commits among those databases that support this protocol.

  NONE Does not use any TM to perform two phase commit and does not enforce single updater, multiple reader. When a COMMIT or ROLLBACK statement is executed, individual COMMITs or ROLLBACKs are posted to all databases. If one or more rollbacks fails an error (SQLSTATE 58005) is raised. If one or more commits fails an error (SQLSTATE 40003) is raised.

Any of the above options can be overridden at run time using a special SET CLIENT application programming interface (API). Their current settings can be obtained using the special QUERY CLIENT API. Note that these are not

11. The CONDITIONAL option will not work properly with downlevel servers prior to Version 2. A disconnection will occur in these cases regardless of the presence of WITH HOLD cursors

SQL statements; they are APIs defined in the various host languages and in the Command Line Processor. These are defined in the *Command Reference* and *Administrative API Reference* manuals.

Data Representation Considerations

Different systems represent data in different ways. When data is moved from one system to another, data conversion sometimes must be performed. Products supporting DRDA will automatically perform any necessary conversions at the receiving system. With numeric data, the information needed to perform the conversion is the data type of the data and how that data type is represented by the sending system. With character data, additional information is needed to convert character strings. String conversion depends on both the code page of the data and the operation that is to be performed with that data. Character conversions are performed in accordance with the IBM Character Data Representation Architecture (CDRA). For more information on character conversion, refer to *Character Data Representation Architecture Reference* SC09-1390.

DB2 Federated Systems

This section provides an overview of the elements of a DB2 federated system, an overview of the tasks that administrators and users of the system perform, and explanations of the concepts associated with these tasks.

The Federated Server, Federated Database, and Data Sources

A *DB2 federated system* is a distributed computing system that consists of:

- A DB2 server, called a *federated server*.

  In a DB2 installation, any number of DB2 instances can be configured to function as federated servers.
- Multiple data sources to which the federated server sends queries.

  Each data source consists of an instance of a relational database management system plus the database or databases that the instance supports. The data sources in a DB2 federated system can include Oracle instances and instances of the members of the DB2 family.

  The data sources are semi-autonomous. For example, the federated server can send queries to Oracle data sources at the same time that Oracle applications can access these data sources. A DB2 federated system does not monopolize or restrict access to Oracle or other data sources (beyond integrity and locking constraints).

To end users and client applications, the data sources appear as a single collective database. In actuality, users and applications interface with a database, called the *federated database*, that is within the federated server. To obtain data from data sources, they submit queries in DB2 SQL to the federated database. DB2 then distributes the queries to the appropriate data sources. DB2 also provides access plans for optimizing the queries (in some cases, these plans call for processing the queries at the federated server rather than at the data source). Finally, DB2 collects the requested data and passes it to the users and applications.

Queries submitted from the federated server to data sources must be read-only. To write to a data source (for example, to update a data source table), users and applications must use the data source's own SQL in a special mode called *pass-through*.

Tasks to Perform in a DB2 Federated System

This section introduces concepts that are associated with tasks that users perform to establish and use a federated system. (In this section and those that follow, the term users refers to all types of personnel who work with federated systems; for example, database administrators, application programmers, and end users).

The following list of tasks identifies the types of users who typically perform the tasks. Be aware that other types of users can also perform these tasks. For example, the list indicates that DBAs typically create mappings between authorizations to access the federated database and authorizations to access data sources. But application programmers and end users can also perform this task.

To establish and use a DB2 federated system:

1. The DBA designates a DB2 server as a federated server. Refer to the *Installation and Configuration Supplement* for information about how this is done.
2. Data sources are set up for access:
   a. The DBA connects to the federated database.
   b. The DBA creates a wrapper for each category of data source that is to be included in the federated system. (*Wrappers* are mechanisms by which the federated server interacts with data sources. Refer to “Wrappers and Wrapper Modules” on page 43 for details.)
   c. The DBA supplies the federated server with a description of each data source. (The description is called a *server definition*. Refer to “Server Definitions and Server Options” on page 44 for details.)
   d. If a user's authorization ID used to access the federated database differs from the user's authorization ID used to access a data source, the DBA defines an association between the two authorization IDs. (This association is called a *user mapping*. Refer to “User Mappings and User Options” on page 46 for details.)
   e. If a default mapping between a DB2 data type and a data source data type does not meet user requirements, the DBA modifies the mapping as needed. (A *data type mapping* is a defined association between two

compatible data types—one supported by the federated database and one supported by a data source. Refer to "Data Type Mappings" on page 47 for details.)

f. If a default mapping between a DB2 function and a data source function does not meet user requirements, the DBA modifies the mapping as needed. (A *function mapping* is a defined association between two compatible functions—one supported by the federated database and one supported by a data source. Refer to "Function Mappings, Function Templates, and Function Mapping Options" on page 48 for details.)

g. DBAs and application programmers create nicknames for the data source tables and views that are to be accessed. (A *nickname* is an identifier by which the federated system references a data source table or view. Refer to "Nicknames and Column Options" on page 48 for details.)

h. Optional: If a data source table has no index, the DBA can provide the federated server with the same sort of information that the definition of an actual index would contain. If a data source table has an index that the federated server is unaware of, the DBA can inform the server of the index's existence. In either case, the information that the DBA supplies helps DB2 to optimize queries of table data. (This information is called an *index specification*. Refer to "Index Specifications" on page 49 for details.)

3. Application programmers and end users retrieve information from data sources:
   - Using DB2 SQL, application programmers and end users query tables and views that are referenced by nicknames. (Queries directed to two or more data sources are called *distributed requests*. Refer to "Distributed Requests" on page 50 for details.)

     In processing a query, the federated server can perform operations that are supported by DB2 SQL but not by the data source's SQL. (This capability is called *compensation*. Refer to "Compensation" on page 51 for details.)
   - Application programmers and end users might occasionally submit queries, DML statements, and DDL statements to data sources in the data sources' own SQL. The programmers and users can do this in pass-through mode. (Refer to "Pass-Through" on page 51 for details.)

The following sections discuss the concepts mentioned in this task list in the same order in which they appear in the list. Some of these sections also introduce related concepts.

Wrappers and Wrapper Modules

A wrapper is the mechanism by which the federated server communicates with, and retrieves data from, a data source. To implement a wrapper, the server uses routines stored in a library called a *wrapper module*. These routines allow the server to perform operations such as connecting to a data source and retrieving data from it iteratively.

There are three wrappers:

- A wrapper with the default name of DRDA is used for all DB2 family data sources.
- A wrapper with the default name of SQLNET is used for all Oracle data sources supported by Oracle's SQL*Net client software.
- A wrapper with the default name of NET8 is used for all Oracle data sources supported by Oracle's Net8 client software.

A wrapper is registered to the federated server with the CREATE WRAPPER statement. Refer to "CREATE WRAPPER" on page 839 for details.

Server Definitions and Server Options

After the DBA registers a wrapper that allows the federated server to interact with data sources, the DBA defines those data sources to the federated database. This section:

- Describes the definition that the DBA provides
- Describes the SQL for specifying certain parameters, called *server options*, that contain portions of a server definition
- Distinguishes different meanings of the term "server".

Introduction to Server Definitions

In defining a data source to the federated database, the DBA supplies a name for the data source as well as information that pertains to the data source. This information includes the type and version of the RDBMS of which the data source is an instance, and the RDBMS's name for the data source. It also includes metadata that is specific to the RDBMS. For example, a DB2 family data source can have multiple databases, and the definition of such a data source must specify which database the federated server can connect to. In contrast, an Oracle data source has one database, and the federated server can connect to the database without needing to know its name. The name is therefore not included in the federated server's definition of the data source.

The name and information that the DBA supplies is collectively called a server definition. This term reflects the fact that data sources answer requests for data and are therefore servers in their own right. Other terms reflect this fact also. For example:

- Some of the information within a server definition is stored as server options. Thus, the name for a data source is stored as a value of a server option called NODE. For a DB2 family data source, the name of the database to which the federated server connects is stored as a value of a server option called DBNAME.

- The SQL statements for creating and modifying a server definition are called CREATE SERVER and ALTER SERVER, respectively.

SQL Statements for Setting Server Options

Values are assigned to server options through the CREATE SERVER, ALTER SERVER, and SET SERVER OPTION statements.

The CREATE SERVER and ALTER SERVER statements set server options to values that persist over successive connections to the data source. These values are stored in the catalog. Consider this scenario: A federated system DBA uses the CREATE SERVER statement to define a new Oracle data source to the federated system. This data source's database uses the same collating sequence that the federated database uses. The DBA wants the optimizer to know about this match, so that the optimizer can take advantage of it to expedite performance. Accordingly, in the CREATE SERVER statement, the DBA sets a server option called COLLATING_SEQUENCE to 'Y' (yes, the collating sequences at the data source and federated database are the same). The setting of 'Y' is recorded in the catalog, and it remains in effect while users and applications access the Oracle data source.

Some months later, the Oracle DBA changes the Oracle data source's collating sequence. Therefore, the federated system DBA resets COLLATING_SEQUENCE to 'N' (no, the data source's collating sequence is not the same as the federated database's). The DBA uses the ALTER SERVER statement to make this update. The catalog is updated also, and the new setting remains in effect as users and applications continue to access the data source.

The SET SERVER OPTION statement overrides a server option value temporarily, for the duration of a single connection to the federated database. The overriding value does not get stored in the catalog.

To illustrate: A server option called PLAN_HINTS can be set to a value that enables DB2 to supply Oracle data sources with statement fragments, called plan hints, that help Oracle optimizers to do their job. For example, plan hints can help an optimizer to decide what index to use in accessing a table, or what table join sequence to use in retrieving data for a result set.

For data sources ORACLE1 and ORACLE2, the PLAN_HINTS server option is set to its default, 'N' (no, do not furnish these data sources with plan hints). Then a programmer writes a distributed request for data from ORACLE1 and ORACLE2; and the programmer expects that plan hints would help the optimizers at these data sources to improve their strategies for accessing this data. Accordingly, the programmer uses the SET SERVER OPTION statement to override the 'N' with 'Y' (yes, furnish plan hints). The 'Y' stays in effect only while the application that contains the request is connected to the federated database; it does not get stored in the catalog.

Refer to "CREATE SERVER" on page 708, "ALTER SERVER" on page 473, and "SET SERVER OPTION" on page 1035 for more information. Refer to "Server Options" on page 1249 for descriptions of all server options and their settings.

Three Meanings for "Server"

In the terms *server definition* and *server option*, and in the SQL statements discussed in the preceding section, the word *server* refers to data sources only. It does not refer to the federated server, or to DB2 application servers.

The concepts of DB2 application servers and federated servers, however, overlap. As indicated in "Distributed Relational Database" on page 29, an application server is a database manager instance to which application processes connect and submit requests. This is also true of a federated server; thus, a federated server is a type of application server. But two main things distinguish it from other application servers:

- It is configured to receive requests that are ultimately intended for data sources; and it distributes these requests to the data sources.
- Like other application servers, a federated server uses DRDA communication protocols to communicate with DB2 family instances. Unlike other application servers, a federated server uses sqlnet and net8 communication protocols to communicate with Oracle instances.

User Mappings and User Options

The federated server can send the distributed request of an authorized user or application to a data source under either of these conditions:

- The user or application uses the same user ID for both the federated database and the data source. In addition, if the data source requires a password, the user or application uses the same password for the federated database and the data source.
- The user's or application's authorization to access the federated database differs in some way from the user's or application's authorization to access the data source. In addition, when the user or application requests access to the data source, the federated database authorization is changed to the data source authorization, so that the access can be granted. This change can occur only if a defined association, called a user mapping, exists between the two authorizations.

User mappings can be defined and modified with the CREATE USER MAPPING and ALTER USER MAPPING statements. These statements include parameters, called *user options*, to which values related to authorization are assigned. For example, suppose that a user has the same ID, but different passwords, for the federated database and a data source. For the user to access the data source, it is necessary to map the passwords to one another.

This can be done with a CREATE USER MAPPING statement in which the password at the data source is assigned as a value to a user option called REMOTE_PASSWORD.

Refer to “CREATE USER MAPPING” on page 821, “ALTER USER MAPPING” on page 516, and *Administration Guide* for more information. Refer to “User Options” on page 1254 for descriptions of the user options and their settings.

Data Type Mappings

For the federated server to retrieve data from columns of data source tables and views, the columns’ data types at the data source must map to corresponding data types that are already defined to the federated database. DB2 supplies default mappings for most kinds of data types. For example, the Oracle type FLOAT maps by default to the DB2 type DOUBLE, and the DB2 Universal Database for OS/390 type DATE maps by default to the DB2 type DATE. There are no mappings for the data types that DB2 federated servers do not support: LONG VARCHAR, LONG VARGRAPHIC, DATALINK, large object (LOB) types, and user-defined types.

Refer to “Default Data Type Mappings” on page 1254 for listings of the default data type mappings.

When values from a data source column are returned, they conform fully to the DB2 type in the type mapping that applies to the column. If this mapping is a default, the values also conform fully to the data source type in the mapping. For example, when an Oracle table with a FLOAT column is defined to the federated database, the default mapping of Oracle FLOAT to DB2 DOUBLE will, unless it has been overridden, automatically apply to that column. Consequently, the values returned from the column will conform fully to both FLOAT and DOUBLE.

It is possible to change the format or length of values returned by changing the DB2 type that the values must conform to. For example, the Oracle type DATE is for time stamps. By default, it maps to the DB2 type TIMESTAMP. Suppose that several Oracle table columns have a data type of DATE, and that a user wants queries of these columns to yield times only. The user could then map the Oracle type DATE to the DB2 type TIME, overriding the default. That way, when the columns are queried, only the time portion of the time stamps would be returned.

The CREATE TYPE MAPPING statement can be used to create a modified data type mapping that applies to one or more data sources. The ALTER NICKNAME statement can be used to modify a data type mapping for a specific column of a specific table.

Refer to “CREATE TYPE MAPPING” on page 816, “ALTER NICKNAME” on page 466 , and the *Application Development Guide* for more information.

Function Mappings, Function Templates, and Function Mapping Options

For the federated server to recognize a data source function, there needs to be a mapping between this function and a corresponding DB2 function that already exists at the server. DB2 supplies default mappings between existing built-in data source functions and built-in DB2 functions. If a user wants to use a data source function that the federated server does not recognize—for example, a new built-in function or a user-defined function—then the user must create a mapping between this function and a counterpart at the federated database. If a counterpart does not exist, the user must create one that meets the following requirements:

- If the data source function has input parameters, the counterpart must have the same number of input parameters that the data source function has. If the data source function has no input parameters, the counterpart cannot have any.
- The counterpart’s data types for input parameters (if any) and returned values must be compatible with the data source function’s corresponding data types.

The counterpart can be either a complete function or a function template. A *function template* is a partial function that has no executable code. It cannot be invoked independently; its only purpose is to participate in a mapping with a data source function, so that the data source function can be invoked from the federated server.

Function mappings are created with the CREATE FUNCTION MAPPING statement. This statement includes parameters, called *function mapping options*, to which the user can assign values that pertain to the mapping being created or to the data source function within the mapping. Such values, for example, can include estimated statistics on the overhead that would be consumed when the data source function is invoked. The optimizer uses these estimates in developing strategies for invoking the function.

Refer to “CREATE FUNCTION (Source or Template)” on page 639 and “CREATE FUNCTION MAPPING” on page 657 for details about creating function templates and function mappings. Refer to “Function Mapping Options” on page 1248 for descriptions of the function mapping options and their values. Refer to the *Application Development Guide* for guidelines on optimizing the invocation of data source functions.

Nicknames and Column Options

When a client application submits a distributed request to the federated server, the server parcels out the request to the appropriate data sources. The request does not need to specify these data sources. Instead, it references data source tables and views by nicknames, that map to the tables' and views' names at the data source. The mappings obviate the need to qualify the nicknames by data source names. The locations of the tables and views are transparent to the client application.

Nicknames are not alternate names for tables and views in the same way that aliases are; they are pointers by which the federated server references these objects. Nicknames are defined with the CREATE NICKNAME statement. Refer to "CREATE NICKNAME" on page 681 for details.

When a nickname is created for a table or view, the catalog is populated with metadata that the optimizer can use to facilitate access to the table or view. For example, the catalog is supplied with the names of the DB2 data types to which the data types of the table's or view's columns map. If the nickname is for a table with an index, the catalog is supplied also with information related to the index; for example, the name of each column in the index key.

After a nickname is created, the user can supply the catalog with more metadata for the optimizer; for example, metadata that describes values in certain columns of the table or view that the nickname references. The user assigns this metadata to parameters called *column options*. To illustrate: If a table column contains numeric strings only, the user can indicate this by assigning the value 'Y' to a column option called NUMERIC_STRING. As a result, the optimizer can form strategies to have these strings sorted at the data source, thereby saving the overhead of porting them to the federated server and sorting them there. The savings is especially great when the database that contains the values has a collating sequence that differs from the federated database's collating sequence.

Column options are defined with the ALTER NICKNAME statement. Refer to "ALTER NICKNAME" on page 466 for more information about this statement. Refer to "Column Options" on page 1247 for descriptions of the column options and their settings.

Index Specifications

When a nickname is created for a data source table, the federated server supplies the catalog with information about any indexes that a data source table has. The optimizer uses this information to facilitate retrieval of the table's data. If the table has no indexes, the user can nevertheless supply information that an index definition typically contains; for example, which column or columns in the table to search in order to find information quickly. The user would do this by running a CREATE INDEX statement that contains the information and references the table's nickname.

The user can supply the optimizer with similar information for tables that have indexes of which the federated server is unaware. For example, suppose that nickname NICK1 is created for a table that has no index but that acquires one later, or that nickname NICK2 is created for a view over a table that has an index. In these situations, the federated server would be unaware of the indexes. But the user could use CREATE INDEX statements to inform the server that the indexes exist. One statement would reference NICK1 and contain information about the index of the table that NICK1 identifies. The other would reference NICK2 and contain information about the index of the base table that underlies the view that NICK2 identifies.

In cases such as those just described, the information in the CREATE INDEX statement is cataloged as a set of metadata called an index specification. Be aware that when the statement references a nickname, it produces only an index specification, not an actual index. Refer to "CREATE INDEX" on page 662 and the *Administration Guide: Performance* for more information.

Distributed Requests

A distributed request can use devices such as subqueries and join subselects to specify what table or view columns are be accessed, and what data is to be retrieved.

This section provides examples within the context of the following scenario: A federated server is configured to access a DB2 Universal Database for OS/390 data source, a DB2 Universal Database for AS/400 data source, and an Oracle data source. Stored in each data source is a table that contains employee information. The federated server references these tables by nicknames that refer to where the tables reside: UDB390_EMPLOYEES, AS400_EMPLOYEES, and ORA_EMPLOYEES. In addition to its table of employee information, the Oracle data source has a table that contains information about the countries that the employees live in. The nickname for this second table is ORA_COUNTRIES.

A Request with a Subquery

Table AS400_EMPLOYEES contains the phone numbers of employees who live in Asia. It also contains the country codes associated with these phone numbers, but it doesn't list the countries that the codes represent. Table ORA_COUNTRIES, however, does list both codes and countries. The following query uses a subquery to find out the country code for China; and it uses SELECT and WHERE clauses to list those employees in AS400_EMPLOYEES whose phone numbers require this particular code.

```
SELECT NAME, TELEPHONE
  FROM DJADMIN.AS400_EMPLOYEES
  WHERE COUNTRY_CODE IN
  (SELECT COUNTRY_CODE
     FROM DJADMIN.ORA_COUNTRIES
     WHERE COUNTRY_NAME = 'CHINA')
```

When a distributed request such as the one above is compiled, the compiler's query rewrite facility transforms it into a form that can be optimized more easily.

A Request for a Join

A relational join produces a result set that contains a combination of columns retrieved from two or more tables. Conditions should always be specified to limit the size of the result set's rows.

The query below combines employee names and their corresponding country names by comparing the country codes listed in two tables. Each table resides in a different data source.

```
SELECT T1.NAME, T2.COUNTRY_NAME
  FROM DJADMIN.UDB390_EMPLOYEES T1, DJADMIN.ORA_COUNTRIES T2
  WHERE T1.COUNTRY_CODE = T2.COUNTRY_CODE
```

Compensation

Compensation is the processing of SQL statements for RDBMSs that do not support those statements. Each type of RDBMS (DB2 Universal Database for AS/400, DB2 Universal Database for OS/390, Oracle, and so on) supports a subset of the international standard of SQL. In addition, some types support SQL constructs that exceed this standard. The totality of SQL that a type of RDBMS supports is called an *SQL dialect*. If an SQL construct is found in DB2's SQL dialect, but not in a data source's dialect, the federated server can implement this construct on behalf of the data source.

Example 1: DB2's SQL includes the clause, common-table-expression. In this clause, a name can be specified by which all FROM clauses in a fullselect can reference a result set. The federated server will process a common-table-expression for an Oracle database, even though Oracle's SQL dialect does not include common-table-expression.

Example 2: When connecting to a data source that does not support multiple open cursors within an application, the federated server can simulate this function by establishing separate, simultaneous connections to the data source. Similarly, the federated server can simulate CURSOR WITH HOLD capability for a data source that does not provide that function.

Compensation makes it possible to use DB2's SQL dialect to make all queries supported by the federated server. It is not necessary to use dialects specific to RDBMSs other than DB2.

Pass-Through

Users can use the pass-through function to communicate with data sources in the data sources' own SQL dialect. In pass-through, users can submit not only queries, but also DML and DDL statements. Refer to "SQL Processing in

Pass-Through Sessions" on page 1256 for information on how DB2 and data sources manage the processing of statements submitted in pass-through sessions.

The federated server provides the following SQL statements to manage pass-through sessions:

SET PASSTHRU
: Opens and terminates pass-through sessions.

GRANT (Server Privileges)
: Grants a user, group, list of authorization IDs, or PUBLIC the authority to initiate pass-through sessions to a specific data source.

REVOKE (Server Privileges)
: Revokes the authority to initiate pass-through sessions.

There are certain restrictions on using pass-through. For example, in a pass-through session, a cursor cannot be opened directly against a data source object. Refer to "Considerations and Restrictions" on page 1257 for a complete list of restrictions.

Character Conversion

A *string* is a sequence of bytes that may represent characters. Within a string, all the characters are represented by a common coding representation. In some cases, it might be necessary to convert these characters to a different coding representation. The process of conversion is known as *character conversion*. [12]

Character conversion can occur when an SQL statement is executed remotely. Consider, for example, these two cases:
- The values of host variables sent from the application requester to the application server
- The values of result columns sent from the application server to the application requester.

In either case, the string could have a different representation at the sending and receiving systems. Conversion can also occur during string operations on the same system.

The following list defines some of the terms used when discussing character conversion.

12. Character conversion, when required, is automatic and is transparent to the application when it is successful. A knowledge of conversion is therefore unnecessary when all the strings involved in a statement's execution are represented in the same way. This is frequently the case for *stand-alone* installations and for networks within the same country. Thus, for many readers, character conversion may be irrelevant.

character set	A defined set of characters. For example, the following character set appears in several code pages: • 26 non-accented letters A through Z • 26 non-accented letters a through z • digits 0 through 9 • . , : ; ? ( ) ' " / − _ & + % * = < >
code page	A set of assignments of characters to code points. In the ASCII encoding scheme for code page 850, for example, "A" is assigned code point X'41' and "B" is assigned code point X'42'. Within a code page, each code point has only one specific meaning. A code page is an attribute of the database. When an application program connects to the database, the database manager determines the code page of the application.
code point	A unique bit pattern that represents a character.
encoding scheme	A set of rules used to represent character data. For example: • Single-Byte ASCII • Single-Byte EBCDIC • Double-Byte ASCII • Mixed Single- and Double-Byte ASCII.

Character Sets and Code Pages

The following example shows how a typical character set might map to different code points in two different code pages.

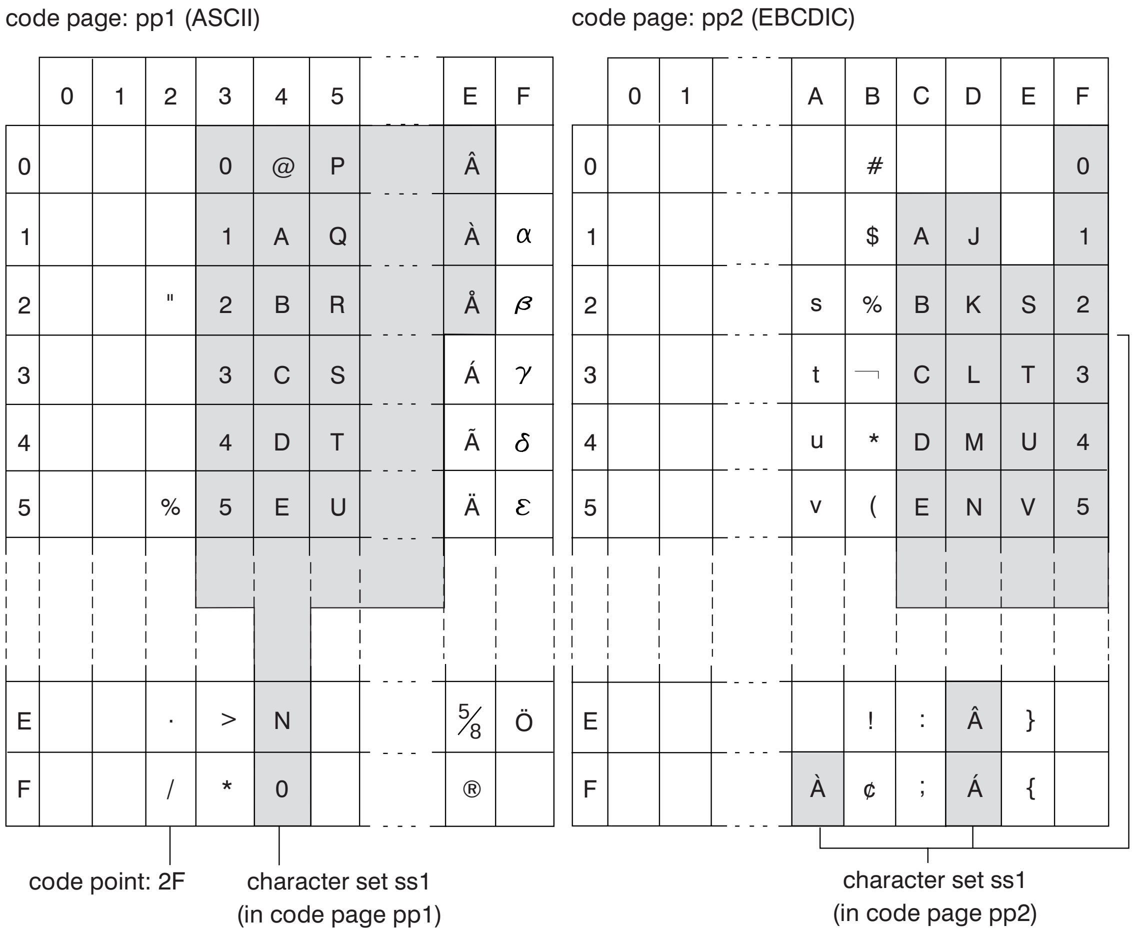


Figure 7. Mapping A Character Set In Different Code Pages

Even with the same encoding scheme, there are many different code pages, and the same code point can represent a different character in different code pages. Furthermore, a byte in a character string does not necessarily represent a character from a single-byte character set (SBCS). Character strings are also used for mixed and bit data. *Mixed data* is a mixture of single-byte, double-byte, or multi-byte characters. *Bit data* (columns defined as FOR BIT DATA or BLOBs, or binary strings) is not associated with any character set.

Code Page Attributes

The database manager determines code page attributes for all character strings when an application is bound to a database. The potential code page attributes are:

The Database Code Page The database code page stored in the database

	configuration files. This code page value is determined when the database is created and cannot be altered.
The Application Code Page	The code page under which the application is executed. Note that this is not necessarily the same code page under which the application was bound. (See the *Application Development Guide* for further information on binding and executing application programs.)
Code Page 0	This represents a string that is derived from an expression that contains a FOR BIT DATA or BLOB value.

String Code Page Attributes

Character string code page attributes are as follows:

- Columns may be in the database code page or code page 0 (if defined as character FOR BIT DATA or BLOB).
- Constants and special registers (for example, USER, CURRENT SERVER) are in the database code page. Note that constants are converted to the database code page when an SQL statement is bound to the database.
- Input host variables are in the application code page.

A set of rules is used to determine the code page attributes for operations that combine string objects, such as the results of scalar operations, concatenation, or set operations. At execution time, code page attributes are used to determine any requirements for code page conversions of strings.

For more details on character conversion, see:

- “Conversion Rules for String Assignments” on page 97 for rules on string assignments
- “Rules for String Conversions” on page 111 for rules on conversions when comparing or combining character strings.

Authorization and Privileges

An *authorization* allows a user or group to perform a general task such as connecting to a database, creating tables, or administering a system. A *privilege* gives a user or group the right to access one specific database object in a specified way.

The database manager requires that a user be specifically authorized, either implicitly or explicitly, [13] to use each database function needed by that user to perform a specific task. Thus to create a table, a user must be authorized to create tables; to alter a table, a user must be authorized to alter the table; and so on.

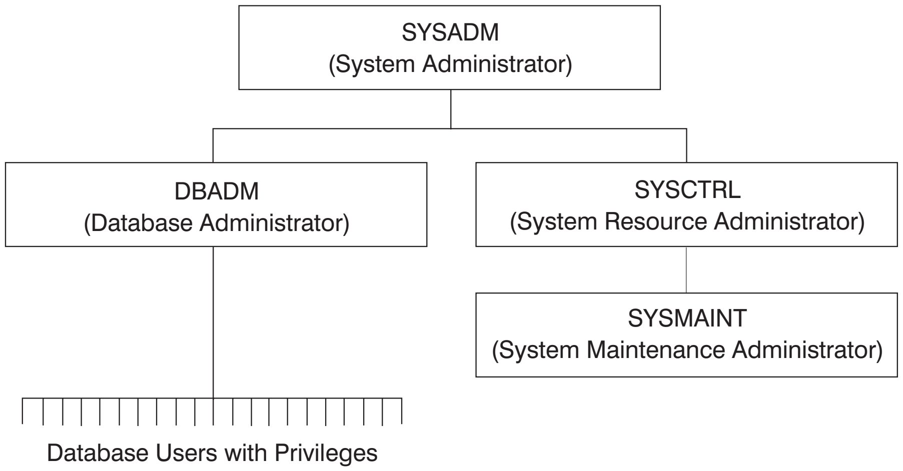

SYSADM
(System Administrator)
DBADM
(Database Administrator)
SYSCTRL
(System Resource Administrator)
SYSMAINT
(System Maintenance Administrator)
Database Users with Privileges

Figure 8. Hierarchy of Authorities and Privileges

The person or persons with administrative authority have the task of controlling the database manager and are responsible for the safety and integrity of the data. They control who will have access to the database manager and to what extent each user has access.

The database manager provides two administrative authorities:

SYSADM
System administrator authority

DBADM
Database administrator authority

and two system control authorities:
SYSCTRL
System control authority
SYSMAINT
System maintenance authority

13. Explicit authorities or privileges are granted to the user (GRANTEETYPE of U). Implicit authorities or privileges are granted to a group to which the user belongs (GRANTEETYPE of G).

SYSADM authority is the highest level of authority and has control over all the resources created and maintained by the database manager. SYSADM authority includes all the privileges of DBADM, SYSCTRL, and SYSMAINT, and the authority to grant or revoke DBADM authorities.

DBADM authority is the administrative authority specific to a single database. This authority includes privileges to create objects, issue database commands, and access the data in any of its tables through SQL statements. DBADM authority also includes the authority to grant or revoke CONTROL and individual privileges.

SYSCTRL authority is the higher level of system control authority and applies only to operations affecting system resources. It does not allow direct access to data. This authority includes privileges to create, update, or drop a database; quiesce an instance or database; and drop or create a table space.

SYSMAINT authority is the second level of system control authority. A user with SYSMAINT authority can perform maintenance operations on all databases associated with an instance. It does not allow direct access to data. This authority includes privileges to update database configuration files, backup a database or table space, restore an existing database, and monitor a database.

Database authorities apply to those activities that an administrator has allowed a user to perform within the database that do not apply to a specific instance of a database object. For example, a user may be granted the authority to create packages but not create tables.

Privileges apply to those activities that an administrator or object owner has allowed a user to perform on database objects. Users with privileges can create objects, though they face some constraints, unlike a user with an authority like SYSADM or DBADM. For example, a user may have the privilege to create a view on a table but not a trigger on the same table. Users with privileges have access to the objects they own, and can pass on privileges on their own objects to other users by using the GRANT statement.

CONTROL privilege allows the user to access a specific database object as desired and to GRANT and REVOKE privileges to and from other users on that object. DBADM authority is required to grant CONTROL privilege.

Individual privileges and database authorities allow a specific function but do not include the right to grant the same privileges or authorities to other users. The right to grant table, view or schema privileges to others can be extended to other users using the WITH GRANT OPTION on the GRANT statement.

Table Spaces and Other Storage Structures

Storage structures contain the objects of the database. The basic storage structures managed by the database manager are table spaces. A *table space* is a storage structure containing tables, indexes, large objects, and data defined with a LONG data type. There are two types of table spaces:

Database Managed Space (DMS) Table Space
A table space which has its space managed by the database manager.

System Managed Space (SMS) Table Space
A table space which has its space managed by the operating system.

All table spaces consist of containers. A *container* describes where objects, such as some tables, are stored. For example, a subdirectory in a file system could be a container.

For more information on table spaces and containers, see "CREATE TABLESPACE" on page 764 or the *Administration Guide*.

Data that is read from table space containers is placed in an area of memory called a *buffer pool*. A buffer pool is associated with a table space allowing control over which data shares the same memory areas for data buffering. For more information on buffer pools, see "CREATE BUFFERPOOL" on page 569 or the *Administration Guide*.

A partitioned database allows data to be spread across different database partitions. The partitions included are determined by the *nodegroup* assigned to the table space. A nodegroup is a group of one or more partitions that are defined as part of the database. A table space includes one or more containers for each partition in the nodegroup. A *partitioning map* is associated with each nodegroup. The partitioning map is used by the database manager to determine which partition from the nodegroup will store a given row of data. For more information on nodegroups and data partitioning, see "Data Partitioning Across Multiple Partitions" on page 59, "CREATE NODEGROUP" on page 684 or the *Administration Guide*.

A table can also include columns that register links to data stored in external files. The mechanism for this is the DATALINK data type. A DATALINK value which is recorded in a regular table points to a file stored in an external file server.

The DB2 Data Links Manager, which is installed on a fileserver, works in conjunction with DB2 to provide the following optional functionality:

- Referential integrity to insure that files currently linked to DB2 are not deleted or renamed.

- Security to insure that only those with suitable SQL privileges on the DATALINK column can read the files linked to that column.
- Coordinated backup and recovery of the file.

The DB2 Data Links Manager product comprises the following facilities:

Data Links File Manager
Registers all the files in a particular file server that are linked to DB2.

Data Links Filesystem Filter
Filters file system commands to insure that registered files are not deleted or renamed. Optionally also filters commands to insure that proper access authority exists.

For more information on DB2 Data Links Manager refer to *Administration Guide*.

Data Partitioning Across Multiple Partitions

DB2 allows great flexibility in spreading data across multiple partitions (nodes) of a partitioned database. Users can choose how to partition their data by declaring partitioning keys and can determine which and how many partitions their table data can be spread across by selecting the nodegroup and table space in which the data should be stored. In addition, a partitioning map (which can be user-updatable) specifies the mapping of partitioning key values to partitions. This makes it possible for flexible workload parallelization across a partitioned database for large tables, while allowing smaller tables to be stored on one or a small number of partitions if the application designer chooses. Each local partition may have local indexes on the data it stores in order to provide high performance local data access.

A partitioned database supports a partitioned storage model, in which the partitioning key is used to partition table data across a set of database partitions. Index data is also partitioned with its corresponding tables, and stored locally at each partition.

Before partitions can be used to store database data, they must be defined to the database manager. Partitions are defined in a file called `db2nodes.cfg`. See the *Administration Guide* for more details about defining partitions.

The partitioning key for a table in a table space on a partitioned nodegroup is specified in the CREATE TABLE statement (or ALTER TABLE statement). If not specified, a partitioning key for a table is created by default from the first column of the primary key. If no primary key is specified, the default partitioning key is the first column defined in that table that has a data type other than a LONG or LOB data type. Partitioned tables must have at least one column that is neither a LONG nor a LOB data type. A table in a table space on a single-partition nodegroup will only have a partitioning key if it is explicitly specified.

Hash partitioning is used to place a row on a partition as follows.

1. A hashing algorithm (partitioning function) is applied to all of the columns of the partitioning key, which results in a partitioning map index being generated.
2. This partitioning map index is used as an index into the partitioning map. The partition number at that index in the partitioning map is the partition where the row is stored.
3. Partitioning maps are associated with nodegroups, and tables are created in table spaces which are on nodegroups.

DB2 supports *partial declustering*, which means that the table can be partitioned across a subset of partitions in the system (that is, a nodegroup). Tables do not have to be partitioned across all the partitions in the system.

Partitioning Maps

Each nodegroup is associated with a *partitioning map*, which is an array of 4 096 partition numbers. The partitioning map index produced by the partitioning function for each row of a table is used as an index into the partitioning map to determine partition on which a row is stored.

Figure 9 on page 61 shows how the row with the partitioning key value (c1, c2, c3) is mapped to partitioning map index 2, which, in turn, references partition p5.

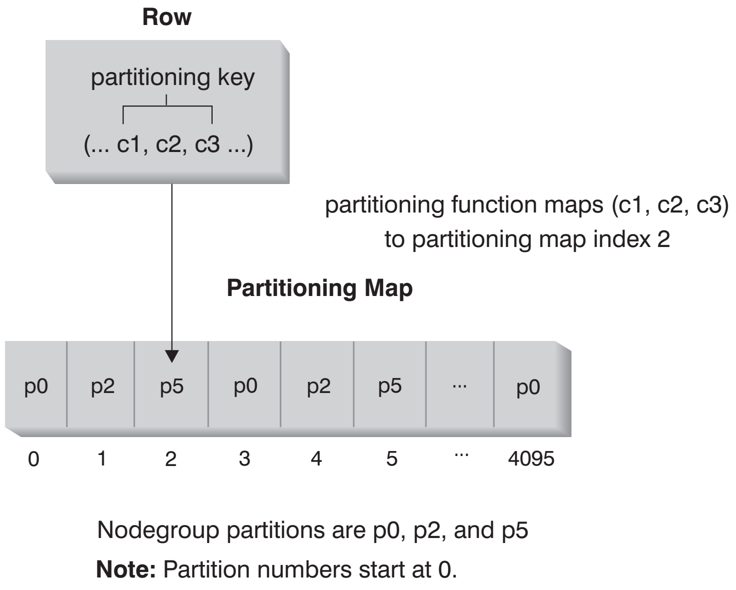

Row
partitioning key
(... c1, c2, c3 ...)
partitioning function maps (c1, c2, c3)
to partitioning map index 2
Partitioning Map
p0 p2 p5 p0 p2 p5 ... p0
0 1 2 3 4 5 ... 4095
Nodegroup partitions are p0, p2, and p5
Note: Partition numbers start at 0.

Figure 9. Data Distribution

The partitioning map can be changed, allowing the data distribution to be changed without modifying the partitioning key or the actual data. The new partitioning map is specified as part of the REDISTRIBUTE NODEGROUP command or API which uses it to redistribute the tables in the nodegroup. See *Command Reference* or *Administrative API Reference* for further information.

Table Collocation

DB2 has the capability of recognizing when the data accessed for a join or subquery is located at the same partition in the same nodegroup. When this happens DB2 can choose to perform the join or subquery processing at the partition where the data is stored, which often has significant performance advantages. This situation is called table collocation. To be considered collocated tables, the tables must:

- be in the same nodegroup (that is not being redistributed [14])
- have partitioning keys with the same number of columns
- have the corresponding columns of the partitioning key be partition compatible (see “Partition Compatibility” on page 114).

OR

- be in a single partition nodegroup defined on the same partition.

14. While redistributing a nodegroup, tables in the nodegroup may be using different partitioning maps - they are not collocated.

Rows in collocated tables with the same partitioning key values will be located on the same partition.