Data Manipulation

Twenty-six percent (26%) of the DB2 UDB V8.1 Family Application Development exam (Exam 703) is designed to test your knowledge of the Structured Query Language (SQL) statements used to manipulate data and to evaluate your understanding of how transactions are used to define points of consistency as data is being manipulated. The questions that make up this portion of the exam are intended to evaluate the following:

➤ Your ability to identify the Data Manipulation Language (DML) statements that are available with DB2 UDB.
➤ Your ability to perform insert, update, and delete operations against a database.
➤ Your ability to retrieve and format data using various forms of the SELECT SQL statement.
➤ Your ability to use DB2 SQL functions.
➤ Your ability to use common table expressions.
➤ Your knowledge of cursors, including the types of cursors available, and the scope in which a cursor can be used once it has been created.
➤ Your ability to identify when cursors should be used in an application.
➤ Your ability to initiate, terminate, and manage transactions.

This chapter is designed to introduce you to the SQL statements that are used to manipulate data and to show you the various ways in which queries can be constructed. This chapter is also designed to show you how data stored in a result data set (produced in response to a query) can be retrieved within an
application program and to provide you with information on how transactions are used to define points of consistency as data is manipulated.

Terms you will learn:

Structured Query Language (SQL)
Data Control Language (DCL) Statements
Data Definition Language (DDL) Statements
Data Manipulation Language (DML) Statements
Transaction Management Statements
INSERT
Subquery
UPDATE
Cursor
DELETE
SELECT
Query
The DISTINCT Clause
The FROM Clause
The WHERE Clause
Relational Predicates
The BETWEEN Predicate
The LIKE Predicate
Wild Card Characters
The IN Predicate
The EXISTS Predicate
The NULL Predicate
The GROUP BY Clause
GROUP BY ROLLUP
GROUP BY CUBE
The HAVING Clause
The ORDER BY Clause
The FETCH FIRST Clause
Cartesian Product
Inner Join
Left Outer Join
Right Outer Join
Full Outer Join
Set Operator
The UNION Set Operator
The UNION ALL Set Operator
The EXCEPT Set Operator
The EXCEPT ALL Set Operator
The INTERSECT Set Operator
The INTERSECT ALL Set Operator

SQL Functions
Cursors
DECLARE CURSOR
OPEN
FETCH
CLOSE
SELECT INTO
VALUES INTO

Transactions
COMMIT
ROLLBACK
Savepoints

Techniques you will master:

Recognizing the various Data Manipulation Language (DML) statements available and understanding how each is used.
Understanding how to add data to a table using the INSERT SQL statement.
Understanding how to modify data stored in a table using the UPDATE SQL statement.
Understanding how to remove data from a table using the DELETE SQL statement.
Knowing how to construct simple and complex queries using the SELECT SQL statement and its clauses.
Knowing how to construct and use common table expressions.
Knowing how to retrieve the results of a query within an application program.
Recognizing the types of cursors available and knowing when each type of cursor should be used.

Understanding how transactions are initiated and terminated and knowing how transaction savepoints are created and used.

Structured Query Language (SQL) Revisited

Earlier, we saw that Structured Query Language (SQL) is a standardized language used to work with database objects and the data they contain. Using SQL, you can define, alter, and remove database objects as well as add, update, delete, and retrieve data values. One of the strengths of SQL is that it can be used in a variety of ways: SQL statements can be executed interactively using tools such as the Command Center and the Command Line Processor (CLP), placed in UNIX shell scripts or Windows batch files for submission to the CLP or some other process, embedded in high-level programming language source code files that are precompiled/compiled to create a database application, or submitted dynamically from applications that do not require precompilation.

Like other languages, SQL has a defined syntax and a set of language elements. Most SQL statements can be categorized according to the functions they have been designed to perform; SQL statements typically fall under one of the following categories:

**Data Control Language (DCL) Statements.** SQL statements used to grant and revoke authorities and privileges.

**Data Definition Language (DDL) Statements.** SQL statements used to create, alter, and delete database objects.

**Data Manipulation Language (DML) Statements.** SQL statements used to store data in and retrieve or remove data from database objects.

**Transaction Management Statements.** SQL statements used to establish and terminate active transactions.

Typically, database administrators use DDL and DCL statements to construct database objects and to control access to those objects once they have been created. Application developers, on the other hand, use, almost exclusively, DML statements. (When developing embedded SQL applications, the Transaction Management Statements are used as well; when developing CLI/ODBC, JDBC, and SQLJ applications, functions and methods that
provide the same functionality are used instead.) With that in mind, this chapter focuses on introducing you to the DML statements available with DB2 UDB.

Although basic syntax is presented for most of the SQL statements covered in this chapter, the actual syntax supported may be much more complex. To view the complete syntax for a specific SQL statement or to obtain more information about a particular statement, refer to the *IBM DB2 Universal Database, Version 8 SQL Reference Volume 2* product documentation.

Data Manipulation Language (DML) Statements

The majority of the work performed by most database applications focuses on data manipulation. Data is often collected from a user via a custom interface and stored in a database; over time, data that has been stored in a database may need to be modified or deleted; and eventually, the need to retrieve specific pieces of data stored in a database arises. To perform these types of operations, database applications rely on Data Manipulation Language (DML) statements. With DB2 UDB (as with most other relational database management systems), four DML statements are available:

- INSERT
- UPDATE
- DELETE
- SELECT

Technically, SELECT is not an actual SQL statement. Instead, it is an informality that is used to define a query (which the ISO SQL standard defines as “an operation on zero or more tables that produces a derived table as a result”). That’s why the syntax for the SELECT statement is covered under Chapter 4, “Queries in the *IBM DB2 Universal Database, Version 8 SQL Reference*” rather than in Chapter 5, “Statements.” (The DB2 UDB documentation also uses the terms “fullselect” and “subselect” to refer to queries.)

The **INSERT** Statement

When a table is first created, it is empty. However, once created, a table can be populated in a variety of ways: It can be bulk-loaded using the LOAD utility, it can be bulk-loaded using the IMPORT utility, or one or more rows can be added to it by executing the **INSERT** SQL statement. Of the three methods available, the **INSERT** statement is the one most commonly used, and it can work directly with the table to be populated or with an updatable
view that references the table to be populated. The basic syntax for the
INSERT statement is:

\[
\text{INSERT INTO } [\text{TableName} | \text{ViewName}] \\
< ( [\text{ColumnName}],...) > \\
\text{VALUES ( [Value],...)}
\]

or

\[
\text{INSERT INTO } [\text{TableName} | \text{ViewName}] \\
< ( [\text{ColumnName}],...) > \\
[\text{SELECTStatement}]
\]

where:

- **TableName** Identifies the name assigned to the table to which data is to be added.
- **ViewName** Identifies the name assigned to the updatable view to which data is to be added.
- **ColumnName** Identifies the name of one or more columns that data values being added to the table/view are to be assigned to. Each name provided must identify an existing column in the table or updatable view specified.
- **Value** Identifies one or more data values that are to be added to the column(s), table, or updatable view specified.
- **SELECTStatement** Identifies a SELECT SQL statement that, when executed, will produce the data values to be added to the column(s), table, or updatable view specified (by retrieving data from other tables and/or views).

So, if you wanted to add a record to a base table named DEPARTMENT that has the following characteristics:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTNO</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DEPTNAME</td>
<td>CHAR(20)</td>
</tr>
<tr>
<td>MGRID</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

you could do so by executing an INSERT statement that looks something like this:

\[
\text{INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, MGRID)} \\
\text{VALUES (001, 'SALES', 1001)}
\]
It is important to note that the number of values provided in the VALUES clause must be equal to the number of column names provided in the column name list. Furthermore, the values provided will be assigned to the columns specified based on the order in which they appear—in other words, the first value provided will be assigned to the first column identified in the column name list, the second value provided will be assigned to the second column identified, and so on. Each value provided must also be compatible with the data type of the column that the value is to be assigned to.

If values are provided for every column found in the table (in the VALUES clause), the column name list can be omitted. In this case, the first value provided will be assigned to the first column found in the table, the second value provided will be assigned to the second column found, and so on. Thus, the row of data that was added to the DEPARTMENT table in the previous example could just as well have been added by executing the following INSERT statement:

```
INSERT INTO DEPARTMENT VALUES (001, 'SALES', 1001)
```

Along with literal values, two keywords can be used to designate values that are to be assigned to base table columns. The first of these is the DEFAULT keyword, which is used to assign a system or user-supplied default value to a column defined with the WITH DEFAULT constraint. The second is the NULL keyword, which is used to assign a NULL value to any column that was not defined with the NOT NULL constraint. (Both of these constraints were covered in detail in Chapter 2, “Database Objects and Programming Methods.”) Thus, you could add a record that contains a NULL value for the MGRID column to the DEPARTMENT table we looked at earlier by executing an INSERT statement that looks something like this:

```
INSERT INTO DEPARTMENT VALUES (001, 'SALES', NULL)
```

By using a special form of the INSERT SQL statement, the results of a query can also be used to provide values for one or more columns in a base table. With this form of the INSERT statement, a SELECT statement (known as a subquery) is provided in place of the VALUES clause (we’ll look at the SELECT statement shortly), and the results of the SELECT statement are assigned to the appropriate columns. (This form of the INSERT statement creates a type of “cut and paste” action in which values are retrieved from one base table or view and inserted into another.) As you might imagine, the number of values returned by the subquery must match the number of columns provided in the column name list (or the number of columns found in the table if no column name list is provided), and the order of assignment is the same as that used when literal values are provided in a VALUES clause. Therefore, using the
results of a query, you could add a record to the DEPARTMENT table we
looked at earlier by executing an INSERT statement that looks something like
this:

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME)
SELECT DEPTNO, DEPTNAME FROM OLD_DEPARTMENT
```

You may have noticed that the INSERT statement used in the last example did not
provide values for every column found in the DEPARTMENT table. Just as
there are times you may want to insert complete records into a table, there may
be times when you wish to insert partial records into a table. Such operations
can be performed by listing only the columns you have data values for in the col-
umn names list and providing the corresponding values using either the VALUES
clause or a subquery. However, for such an INSERT statement to execute cor-
rectly, all columns in the table that the record is being inserted into that do not
appear in the column name list provided must either accept null values or have
a default value constraint defined. Otherwise, the INSERT statement will fail.

Subqueries usually appear within the search condition of a WHERE clause or a
HAVING clause (although subqueries can also be used with insert, update, and
delete operations). A subquery may include search conditions of its own, and
these search conditions may in turn include their own subqueries. When such
“nested” subqueries are processed, the DB2 Database Manager executes the
innermost query first and uses the results to execute the next outer query, and
so on until all nested queries have been processed.

### The UPDATE Statement

Data stored in a database is rarely static; over time, the need to modify (or
even remove) one or more values residing in a database can arise. In such sit-
suations, specific data values can be changed by executing the UPDATE SQL
statement. The basic syntax for this statement is:

```
UPDATE [TableName | ViewName]
SET [(ColumnName) = [Value] | NULL | DEFAULT ,... ]
<WHERE [Condition]>
```

or

```
UPDATE [TableName | ViewName]
SET ([(ColumnName) ,... ) = ([(Value] | NULL | DEFAULT ,... )
<WHERE [Condition]>
```

or

```
UPDATE [TableName | ViewName]
SET ([(ColumnName) ,... ) = ( [SELECTStatement] )
<WHERE [Condition]>
```
where:

- **TableName** Identifies the name assigned to the table that contains the data to be modified.
- **ViewName** Identifies the name assigned to the updatable view that contains the data to be modified.
- **ColumnName** Identifies the name of one or more columns that contain data values to be modified. Each name provided must identify an existing column in the table or updatable view specified.
- **Value** Identifies one or more data values that are to be used to replace existing value(s) found in the column(s) specified.
- **SELECTStatement** Identifies a SELECT SQL statement that, when executed, will produce the data values to be used to replace existing values found in the columns specified (by retrieving data from other tables and/or views).
- **Condition** Identifies the search criterion that is to be used to locate one or more specific rows whose data values are to be modified. (This condition is coded like the WHERE clause that can be used with a SELECT SQL statement; we will look at the WHERE clause and its predicates later.) If no condition is provided, every row found in the table or updatable view specified will be updated.

Thus, if you wanted to modify the records stored in a base table named EMPLOYEE that has the following characteristics:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNO</td>
<td>INTEGER</td>
</tr>
<tr>
<td>FNAME</td>
<td>CHAR(20)</td>
</tr>
<tr>
<td>LNAME</td>
<td>CHAR(30)</td>
</tr>
<tr>
<td>TITLE</td>
<td>CHAR(10)</td>
</tr>
<tr>
<td>DEPARTMENT</td>
<td>CHAR(20)</td>
</tr>
<tr>
<td>SALARY</td>
<td>DECIMAL(6,2)</td>
</tr>
</tbody>
</table>
such that the salary of every employee who has the title of DBA is increased by 10%, you could do so by executing an UPDATE statement that looks something like this:

```sql
UPDATE EMPLOYEE SET SALARY = SALARY * 1.10
WHERE TITLE = 'DBA'
```

The UPDATE statement can also be used to remove values from nullable columns. This is done by changing the column’s current value to NULL. Thus, the value assigned to the DEPARTMENT column of the EMPLOYEE table shown in the previous example could be removed by executing the following UPDATE statement:

```sql
UPDATE EMPLOYEE SET SALARY = NULL
```

Like the INSERT statement, the UPDATE statement can work either directly with the table that contains the values to be modified or with an updatable view that references the table containing the values to be modified. Similarly, the results of a query can be used to provide values for one or more columns identified in the column name list provided. As you might imagine, the number of values returned by the query must match the number of columns provided in the column name list specified. Thus, using the results of a query, you could change the value assigned to the DEPARTMENT columns of each record found in the EMPLOYEE table we looked at earlier by executing an UPDATE statement that looks something like this:

```sql
UPDATE EMPLOYEE SET (DEPARTMENT) =
(SELECT DEPTNAME FROM DEPARTMENT WHERE DEPTNO = 1)
```

It is important to note that updates can be conducted by performing either a searched update or a positioned update operation. So far, all of the examples we have looked at have been searched update operations. To perform a positioned update, a cursor must first be created, opened, and positioned on the row that is to be updated. Then, the UPDATE statement that is to be used to modify one or more data values must contain a WHERE CURRENT OF [CursorName] clause (CursorName identifies the cursor being used—we’ll look at cursors shortly). Because positioned update operations can be conducted only when a valid cursor exists, they can only be performed from within embedded SQL applications.

It is very important that you provide a proper WHERE clause when the UPDATE statement is used. Failure to do so will cause an update operation to be performed on every row found in the specified table or updatable view.

**NOTE**

The **DELETE Statement**

Although the UPDATE statement can be used to delete individual values from a base table (by setting those values to NULL), it cannot be used to remove
entire rows. When one or more rows of data need to be removed from a base table, the **DELETE** SQL statement must be used instead. As with the **INSERT** and **UPDATE** statements, the **DELETE** statement can work either directly with the table that rows are to be removed from or with an updatable view that references the table that rows are to be removed from. The basic syntax for the **DELETE** statement is:

```
DELETE FROM [TableName | ViewName]
<WHERE [Condition]>
```

where:

- **TableName** Identifies the name assigned to the table from which data is to be removed.
- **ViewName** Identifies the name assigned to the deletable view from which data is to be removed.
- **Condition** Identifies the search criterion to be used to locate one or more specific rows that are to be removed. (This condition is coded like the **WHERE** clause used with a **SELECT** SQL statement; we will look at the **WHERE** clause and its predicates later.) If no condition is provided, every row found in the table or deletable view specified will be deleted.

Therefore, if you wanted to remove every record for company XYZ from a base table named SALES that has the following characteristics:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PONUMBER</td>
<td>CHAR(10)</td>
</tr>
<tr>
<td>COMPANY</td>
<td>CHAR(20)</td>
</tr>
<tr>
<td>PURCHASEDATE</td>
<td>DATE</td>
</tr>
<tr>
<td>SALESPERSON</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

you could do so by executing a **DELETE** statement that looks something like this:

```
DELETE FROM SALES WHERE COMPANY = 'XYZ'
```

Like update operations, delete operations can be conducted in one of two ways: as a **searched delete** or as a **positioned delete**. To perform a positioned delete, a cursor must first be created, opened, and positioned on the row to be deleted. Then, the **DELETE** statement to be used to remove the row must contain a **WHERE**
CURRENT of [CursorName] clause (CursorName identifies the cursor being used). Because positioned delete operations can be conducted only when a valid cursor exists, they can only be performed from within embedded SQL applications.

Because omitting the WHERE clause in a DELETE SQL statement causes the delete operation to be applied to all rows in the table or view specified, it is important that you always provide a WHERE clause with a DELETE statement unless you explicitly want to erase all data stored in a table.

The SELECT Statement

Although the primary function of a database is to act as a data repository, sooner or later, almost all database users and/or applications have the need to retrieve specific pieces of information (data) from the database they are interacting with. The operation used to retrieve data from a database is called a query (because it searches or queries the database to find the answer to some question), and the results returned by a query are typically expressed in one of two forms: as a single row of data values or as a set of rows of data values, otherwise known as a result data set (or result set). (If no data values that correspond to the query specification provided can be found in the database, an empty result data set will be returned.)

All queries are created using the SELECT SQL statement, which is an extremely powerful statement that can be used to construct a wide variety of queries containing an infinite number of variations (using a finite set of rules). And because the SELECT statement is recursive, a single SELECT statement can derive its output from a successive number of nested SELECT statements (which are known as subqueries). (We have already seen how SELECT statements can be used to provide input to INSERT and UPDATE statements; SELECT statements can also be used to provide input to other SELECT statements in a similar manner.)

In its simplest form, the syntax for the SELECT statement is:

```
SELECT * FROM [ [TableName] | [ViewName] ]
```

where:

- **TableName** Identifies the name assigned to the table from which data is to be retrieved.
- **ViewName** Identifies the name assigned to the view from which data is to be retrieved.

Consequently, if you wanted to retrieve all values stored in a table or view named DEPARTMENT, you could do so by executing a SELECT statement that looks something like this:

```
SELECT * FROM DEPARTMENT
```
The **SELECT** Statement and Its Clauses

We just saw that if you wanted to retrieve all values stored in a base table, you could do so by executing a **SELECT** statement that looks something like this:

```sql
SELECT * FROM [TableName]
```

But what if you only wanted to see the values stored in two columns of a table? Or what if you wanted the data retrieved to be ordered alphabetically in ascending order? (Data is stored in a table in no particular order, and unless otherwise specified, a query only returns data in the order in which it is found.) How do you construct a query using a **SELECT** SQL statement that retrieves only certain data values and returns those values in a very specific format? You do so by using a more advanced form of the **SELECT** SQL statement to construct your query. The syntax used to construct more advanced forms of the **SELECT** SQL statement is:

```sql
SELECT <DISTINCT> [* | [Expression] <<AS> [NewColumnName]> ,...]
FROM [[TableName] | [ViewName] <<AS> [CorrelationName]> ,...]
<WhereClause>
<GroupByClause>
<HavingClause>
<OrderByClause>
<FetchFirstClause>
```

where:

- **Expression** Identifies one or more columns for which values are to be returned when the **SELECT** statement is executed. The value specified for this option can be any valid SQL language element; however, column names that correspond to the table or view specified in the **FROM** clause are commonly used.

- **NewColumnName** Identifies a new column name to be used in place of the corresponding table or view column name specified in the result data set returned by the **SELECT** statement.

- **TableName** Identifies the name(s) assigned to one or more tables from which data is to be retrieved.
ViewName Identifies the name(s) assigned to one or more views from which data is to be retrieved.

CorrelationName Identifies a shorthand name that can be used when referencing the table or view that the correlation name is associated with in any of the SELECT statement clauses.

WhereClause Identifies a WHERE clause that is to be used with the SELECT statement.

GroupByClause Identifies a GROUP BY clause that is to be used with the SELECT statement.

HavingClause Identifies a HAVING clause that is to be used with the SELECT statement.

OrderByClause Identifies an ORDER BY clause that is to be used with the SELECT statement.

FetchFirstClause Identifies a FETCH FIRST clause that is to be used with the SELECT statement.

If the DISTINCT clause is specified with the SELECT statement, duplicate rows are not repeated in the final result data set returned. (Rows are considered duplicates if the values in all corresponding columns are identical. For the purpose of determining whether values are identical, null values are considered equal.) However, if the DISTINCT clause is used, the result data set produced must not contain columns that hold LONG VARCHAR, LONG VARGRAPHIC, DATALINK, BLOB, CLOB, or DBCLOB data.

So if you wanted to retrieve all values for the columns named WORKDEPT and JOB from a table named EMPLOYEE, you could do so by executing a SELECT statement that looks something like this:

```
SELECT WORKDEPT, JOB FROM EMPLOYEE
```

And when this statement is executed, you might see a result data set that looks something like this:

```
WORKDEPT    JOB
-----------  ----
A00          PRES
B01          MANAGER
C01          MANAGER
E01          MANAGER
D11          MANAGER
D21          MANAGER
E11          MANAGER
E21          MANAGER
```
A00        SALESREP
A00        CLERK
C01        ANALYST
C01        ANALYST
D11        DESIGNER
D11        DESIGNER
D11        DESIGNER
D11        DESIGNER
D11        DESIGNER
D11        DESIGNER
D11        DESIGNER
D11        DESIGNER
D21        CLERK
D21        CLERK
D21        CLERK
D21        CLERK
E11        OPERATOR
E11        OPERATOR
E11        OPERATOR
E11        OPERATOR
E21        FIELDREP
E21        FIELDREP
E21        FIELDREP

32 record(s) selected.

On the other hand, if you wanted to retrieve the same data values but remove all duplicate records found, you could do so by executing the same SELECT statement using the DISTINCT clause. The resulting SELECT statement would look something like this:

```
SELECT DISTINCT WORKDEPT, JOB FROM EMPLOYEE
```

This time, when the SELECT statement is executed you should see a result data set that looks something like this:

```
WORKDEPT   JOB
--------   --------
C01        ANALYST
A00        CLERK
D21        CLERK
D11        DESIGNER
E21        FIELDREP
B01        MANAGER
C01        MANAGER
D11        MANAGER
D21        MANAGER
E01        MANAGER
```
Now suppose you wanted to retrieve all unique values (no duplicates) for the column named JOB from a table named EMPLOYEE and you wanted to change the name of the JOB column in the result data set produced to TITLES. You could do so by executing a SELECT statement that looks something like this:

```
SELECT DISTINCT JOB AS TITLE FROM EMPLOYEE
```

When this statement is executed, you should see a result set that looks something like this:

```
TITLE
--------
ANALYST
CLERK
DESIGNER
FIELDREP
MANAGER
OPERATOR
PRES
SALESREP
```

8 record(s) selected.

You could also produce the same result data set by executing the same SELECT SQL statement using the correlation name “EMP” for the table named EMPLOYEE. The only difference is that the SELECT statement would look something like this:

```
SELECT DISTINCT EMP.JOB AS TITLE FROM EMPLOYEE AS EMP
```

Notice that the column named JOB is qualified with the same correlation name assigned to the table named EMPLOYEE. For this example, this is not really necessary because data is only being retrieved from one table and no two columns in a table can have the same name. However, if data was being retrieved from two or more tables and if columns in different tables had the same name, the qualifier (either the table name or the correlation name) would be needed to tell the DB2 Database Manager which table to retrieve data from for that particular column.
If you were counting when we examined the syntax for the SELECT statement earlier, you may have noticed that a single SELECT statement can contain up to seven different clauses. These clauses are:

➤ The DISTINCT clause
➤ The FROM clause
➤ The WHERE clause
➤ The GROUP BY clause
➤ The HAVING clause
➤ The ORDER BY clause
➤ The FETCH FIRST clause

(Incidentally, these clauses are processed in the order shown.) We saw in the previous SELECT statement examples how the DISTINCT and FROM clauses are used. Now let’s turn our attention to the other clauses that the SELECT statement recognizes.

**The WHERE Clause**

The WHERE clause is used to tell the DB2 Database Manager how to select the rows that are to be returned in the result data set produced in response to a query. When specified, the WHERE clause is followed by a search condition that is essentially a simple test that, when applied to a row of data, will evaluate to “TRUE”, “FALSE”, or “Unknown”. If this test evaluates to “TRUE”, the row is to be returned in the result data set produced; if the test evaluates to “FALSE” or “Unknown”, the row is skipped.

The search condition of a WHERE clause is made up of one or more predicates that are used to compare the contents of a column with a constant value, the contents of a column with the contents of another column from the same table, or the contents of a column in one table with the contents of a column from another table (just to name a few). Some of the more common types of WHERE clause predicates DB2 UDB recognizes include:

➤ Relational predicates (comparisons)
➤ BETWEEN
➤ LIKE
➤ IN
➤ EXISTS
➤ NULL
Each of these predicates can be used alone, or two or more can be combined by using parentheses or logical operators such as AND, OR, and NOT.

**Relational Predicates**

The *relational predicates* (or *comparison operators*) consist of a set of operators that are used to define a comparison relationship between the contents of a column and a constant value, two columns from the same table, or a column in one table with those of a column from another table. The following comparison operators are available:

- \(<\) (Less than)
- \(>\) (Greater than)
- \(\leq\) (Less than or equal to)
- \(\geq\) (Greater than or equal to)
- \(=\) (Equal to)
- \(<\)\(>\) (Not equal to)

Typically, relational predicates are used to include or exclude specific rows from the final result data set produced in response to a query. Thus, if you wanted to retrieve values for the columns named EMPNO and SALARY in a table named EMPLOYEE in which the value for the SALARY column is greater than or equal to $40,000.00, you could do so by executing a `SELECT` statement that looks something like this:

```
SELECT EMPNO, SALARY FROM EMPLOYEE
WHERE SALARY >= 40000.00
```

When this `SELECT` statement is executed, you might see a result data set that looks something like this:

```
EMPNO    SALARY
------   --------
000010   52750.00
000020   41250.00
000050   40175.00
000110   46500.00
```

4 record(s) selected.

It is important to note that the data types of all items involved in a relational predicate comparison must be compatible or the comparison will fail. If necessary, scalar functions can be used (to make the necessary conversions) in conjunction with the relational predicate to meet this requirement. Also keep in mind that all character data is case sensitive. Again, functions are available
that can be used to construct queries that will locate character values, regardless of the case used when they were stored.

**The BETWEEN Predicate**

The BETWEEN predicate is used to define a comparison relationship in which a value is checked to determine whether it falls within a range of values. As with relational predicates, the BETWEEN predicate is used to include or exclude specific rows from the result data set produced in response to a query.

So, if you wanted to retrieve values for the columns named EMPNO and SALARY in a table named EMPLOYEE in which the value for the SALARY column is greater than or equal to $10,000.00 and less than or equal to $20,000.00, you could do so by executing a SELECT statement that looks something like this:

```sql
SELECT EMPNO, SALARY FROM EMPLOYEE
WHERE SALARY BETWEEN 10000.00 AND 20000.00
```

When this SELECT statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000210</td>
<td>18270.00</td>
</tr>
<tr>
<td>000250</td>
<td>19180.00</td>
</tr>
<tr>
<td>000260</td>
<td>17250.00</td>
</tr>
<tr>
<td>000290</td>
<td>15340.00</td>
</tr>
<tr>
<td>000300</td>
<td>17750.00</td>
</tr>
<tr>
<td>000310</td>
<td>15900.00</td>
</tr>
<tr>
<td>000320</td>
<td>19950.00</td>
</tr>
</tbody>
</table>

7 record(s) selected.

If the NOT (negation) operator is used with the BETWEEN predicate (or with any other predicate, for that matter), the meaning of the predicate is reversed. (In the case of the BETWEEN predicate, contents of a column are checked, and only values that fall outside the range of values specified are returned to the final result data set produced.) Thus, if you wanted to retrieve values for the columns named EMPNO and SALARY in a table named EMPLOYEE in which the value for the SALARY column is less than $10,000.00 and more than $30,000.00, you could do so by executing a SELECT statement that looks something like this:

```sql
SELECT EMPNO, SALARY FROM EMPLOYEE
WHERE SALARY NOT BETWEEN 10000.00 AND 30000.00
```
When this *SELECT* statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>52750.00</td>
</tr>
<tr>
<td>000020</td>
<td>41250.00</td>
</tr>
<tr>
<td>000030</td>
<td>38250.00</td>
</tr>
<tr>
<td>000050</td>
<td>40175.00</td>
</tr>
<tr>
<td>000060</td>
<td>32250.00</td>
</tr>
<tr>
<td>000070</td>
<td>36170.00</td>
</tr>
<tr>
<td>000110</td>
<td>46500.00</td>
</tr>
</tbody>
</table>

7 record(s) selected.

**The LIKE Predicate**

The *LIKE* predicate is used to define a comparison relationship in which a character value is checked to see whether it contains a specific pattern of characters. The pattern of characters specified can consist of regular alphanumeric characters and/or special metacharacters that DB2 UDB recognizes, which are interpreted as shown below:

- The underscore character (_ ) is treated as a wild card character that stands for any single alphanumeric character.
- The percent character ( % ) is treated as a wild card character that stands for any sequence of alphanumeric characters.

Thus, if you wanted to retrieve values for the columns named EMPNO and LASTNAME in a table named EMPLOYEE in which the value for the LASTNAME column begins with the letter “S”, you could do so by executing a *SELECT* statement that looks something like this:

```sql
SELECT EMPNO, LASTNAME FROM EMPLOYEE
WHERE LASTNAME LIKE 'S%'
```

When this *SELECT* statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>000060</td>
<td>STERN</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
</tr>
<tr>
<td>000180</td>
<td>SCOUTTEN</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
</tr>
</tbody>
</table>
When using wild card characters, you must take care to ensure that they are placed in the appropriate location in the pattern string specified. Note that in the previous example, only records for employees whose last names begin with the letter “S” are returned. If the character string pattern specified had been “%S%”, records for employees whose last name contains the character “S” (anywhere in the name) would have been returned, and the result data set produced may have looked something like this instead:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
</tr>
<tr>
<td>000020</td>
<td>THOMPSON</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
</tr>
<tr>
<td>000070</td>
<td>PULASKI</td>
</tr>
<tr>
<td>000090</td>
<td>HENDERSON</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
</tr>
<tr>
<td>000110</td>
<td>LUCCHESSEI</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
</tr>
<tr>
<td>000150</td>
<td>ADAMSON</td>
</tr>
<tr>
<td>000170</td>
<td>YOSHIMURA</td>
</tr>
<tr>
<td>000180</td>
<td>SCOUTTEN</td>
</tr>
<tr>
<td>000210</td>
<td>JONES</td>
</tr>
<tr>
<td>000230</td>
<td>JEFFERSON</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
</tr>
<tr>
<td>000260</td>
<td>JOHNSON</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
</tr>
<tr>
<td>000300</td>
<td>SMITH</td>
</tr>
<tr>
<td>000310</td>
<td>SETRIGHT</td>
</tr>
</tbody>
</table>

18 record(s) selected.

Likewise, you must also be careful about using uppercase and lowercase characters in pattern strings; if the data being examined is stored in a case-sensitive manner, the characters used in a pattern string must match the case that was used to store the data in the column being searched, or no corresponding records will be found.

Although the LIKE predicate provides a relatively easy way to search for data values, it should be used with caution; the overhead involved in processing a LIKE predicate can be extremely resource-intensive.
The **IN** Predicate

The **IN** predicate is used to define a comparison relationship in which a value is checked to see whether it matches a value in a finite set of values. This finite set of values can consist of one or more literal values that are coded directly in the **SELECT** statement, or it can be composed of the non-null values found in the result data set generated by a second **SELECT** statement (or subquery).

Thus, if you wanted to retrieve values for the columns named EMPNO and WORKDEPT in a table named EMPLOYEE in which the value for the WORKDEPT column matches a value in a list of department codes, you could do so by executing a **SELECT** statement that looks something like this:

```sql
SELECT LASTNAME, WORKDEPT FROM EMPLOYEE
WHERE WORKDEPT IN ('E11', 'E21')
```

When this **SELECT** statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>WORKDEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>HENDERSON</td>
<td>E11</td>
</tr>
<tr>
<td>SPENSER</td>
<td>E21</td>
</tr>
<tr>
<td>SCHNEIDER</td>
<td>E11</td>
</tr>
<tr>
<td>PARKER</td>
<td>E11</td>
</tr>
<tr>
<td>SMITH</td>
<td>E11</td>
</tr>
<tr>
<td>SETRIGHT</td>
<td>E11</td>
</tr>
<tr>
<td>MEHTA</td>
<td>E21</td>
</tr>
<tr>
<td>LEE</td>
<td>E21</td>
</tr>
<tr>
<td>GOUNOT</td>
<td>E21</td>
</tr>
</tbody>
</table>

9 record(s) selected.

Assuming we don’t know that the values ‘E11’ and ‘E21’ have been assigned to the departments named OPERATIONS and SOFTWARE SUPPORT but we do know that department names and numbers are stored in a table named DEPARTMENT (for normalization) that has two columns named DEPTNO and DEPTNAME, we could produce the same result data set by executing a **SELECT** statement that looks like this:

```sql
SELECT LASTNAME, WORKDEPT FROM EMPLOYEE
WHERE WORKDEPT IN (SELECT DEPTNO FROM DEPARTMENT
                    WHERE DEPTNAME = 'OPERATIONS' OR
                    DEPTNAME = 'SOFTWARE SUPPORT')
```

In this case, the subquery `SELECT DEPTNO FROM DEPARTMENT WHERE DEPTNAME = 'OPERATIONS' OR DEPTNAME = 'SOFTWARE SUPPORT'` produces a result data set that contains the values ‘E11’ and ‘E21’, and the main query evaluates each
value found in the WORKDEPT column of the EMPLOYEE table to determine whether it matches one of the values in the result data set produced by the subquery.

The **exists** Predicate

The `exists` predicate is used to determine whether a particular value exists in a set of rows. The `exists` predicate is always followed by a subquery, and it returns either “TRUE” or “FALSE” to indicate whether a specific value is found in the result data set produced by the subquery. Thus, if you wanted to learn which values found in the column named DEPTNO in a table named DEPARTMENT are used in the column named WORKDEPT found in a table named EMPLOYEE, you could do so by executing a `SELECT` statement that looks something like this:

```
SELECT DEPTNO, DEPTNAME FROM DEPARTMENT
WHERE EXISTS (SELECT WORKDEPT FROM EMPLOYEE
WHERE WORKDEPT = DEPTNO)
```

When this `SELECT` statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>SPIIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>B01</td>
<td>PLANNING</td>
</tr>
<tr>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>D11</td>
<td>MANUFACTURING SYSTEMS</td>
</tr>
<tr>
<td>D21</td>
<td>ADMINISTRATION SYSTEMS</td>
</tr>
<tr>
<td>E01</td>
<td>SUPPORT SERVICES</td>
</tr>
<tr>
<td>E11</td>
<td>OPERATIONS</td>
</tr>
<tr>
<td>E21</td>
<td>SOFTWARE SUPPORT</td>
</tr>
</tbody>
</table>

8 record(s) selected.

In most situations, `exists` predicates are AND-ed with other predicates to determine final row selection.

The **null** Predicate

The `null` predicate is used to determine whether a particular value is a NULL value. Therefore, if you wanted to retrieve values for the columns named FIRSTNME, MIDINIT, and LASTNAME in a table named EMPLOYEE in which the value for the MIDINIT column is a NULL value, you could do so by executing a `SELECT` statement that looks something like this:

```
SELECT FIRSTNME, MIDINIT, LASTNAME FROM EMPLOYEE
WHERE MIDINIT IS NULL
```
When this SELECT statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>FIRSTNAME</th>
<th>MIDINIT</th>
<th>LASTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEAN</td>
<td>-</td>
<td>O'CONNELL</td>
</tr>
<tr>
<td>BRUCE</td>
<td>-</td>
<td>ADAMSON</td>
</tr>
<tr>
<td>DAVID</td>
<td>-</td>
<td>BROWN</td>
</tr>
<tr>
<td>WING</td>
<td>-</td>
<td>LEE</td>
</tr>
</tbody>
</table>

4 record(s) selected.

When using the **NULL** predicate, it is important to keep in mind that null, zero (0), blank (""'), and an empty string (""") are not the same value. Null is a special marker that is used to represent missing information, while zero, blank, and an empty string are actual values that can be stored in a column to indicate a specific value (or lack thereof). Furthermore, some columns accept null values, while other columns do not, depending on their definition. So, before writing SQL statements that check for null values, make sure that the null value is supported by the column(s) being specified.

**The **GROUP BY** Clause**

The **GROUP BY** clause is used to tell the DB2 Database Manager how to organize rows of data returned in the result data set produced in response to a query. In its simplest form, the **GROUP BY** clause is followed by a grouping expression that is usually one or more column names (that correspond to column names found in the result data set to be organized by the **GROUP BY** clause). The **GROUP BY** clause is also used to specify what columns are to be grouped together to provide input to aggregate functions such as **SUM()** and **AVG()**.

Thus, if you wanted to obtain the average salary for all departments found in the column named **DEPTNAME** in a table named **DEPARTMENT** using salary information stored in a table named **EMPLOYEE** and you wanted to organize the data retrieved by department, you could do so by executing a SELECT statement that looks something like this:

```
SELECT DEPTNAME, AVG(SALARY) AS AVG_SALARY
FROM DEPARTMENT D, EMPLOYEE E
WHERE E.WORKDEPT = D.DEPTNO
GROUP BY DEPTNAME
```

When this statement is executed, you might see a result data set that looks something like this:
In this example, each row in the result data set produced contains the department name and the average salary for individuals who work in that department.

A common mistake that is often made when using the \texttt{GROUP BY} clause is the addition of nonaggregate columns to the list of columns that follow the \texttt{GROUP BY} clause. Because grouping is performed by combining all of the nonaggregate columns together into a single concatenated key and breaking whenever that key value changes, extraneous columns can cause unexpected breaks to occur.

\textbf{The \texttt{GROUP BY ROLLUP} Clause}

The \texttt{GROUP BY ROLLUP} clause is used to analyze a collection of data in a single (hierarchal) dimension, but at more than one level of detail. For example, you could group data by successively larger organizational units, such as team, department, and division, or by successively larger geographical units, such as city, county, state or province, country, and continent. Thus, if you were to execute a \texttt{SELECT} statement that looks something like this:

\begin{verbatim}
SELECT WORKDEPT AS DEPARTMENT, AVG(SALARY) AS AVG_SALARY
FROM EMPLOYEE
GROUP BY ROLLUP (WORKDEPT)
\end{verbatim}

you might see a result data set that looks something like this:

\begin{center}
<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>AVERAGE_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>27303.59</td>
</tr>
<tr>
<td>A00</td>
<td>42833.33</td>
</tr>
<tr>
<td>B01</td>
<td>41250.00</td>
</tr>
<tr>
<td>C01</td>
<td>30156.66</td>
</tr>
<tr>
<td>D11</td>
<td>24677.77</td>
</tr>
<tr>
<td>D21</td>
<td>25153.33</td>
</tr>
<tr>
<td>E01</td>
<td>40175.00</td>
</tr>
<tr>
<td>E11</td>
<td>20998.00</td>
</tr>
<tr>
<td>E21</td>
<td>23827.50</td>
</tr>
</tbody>
</table>
\end{center}

9 record(s) selected.
This result data set contains average salary information for all employees found in the table named EMPLOYEE, regardless of which department they work in (the first line in the result data set returned), as well as average salary information for each department available (the remaining lines in the result data set returned).

In this example, only one expression (known as the grouping expression) is specified in the GROUP BY ROLLUP clause (in this case, the grouping expression is WORKDEPT). However, one or more grouping expressions can be specified in a single GROUP BY ROLLUP clause (for example, GROUP BY ROLLUP (DIVISION, WORKDEPT)). When multiple grouping expressions are specified, the DB2 Database Manager groups the data by all grouping expressions used, then by all but the last grouping expression used, and so on. Then, it makes one final grouping that consists of the entire contents of the specified table. In addition, when specifying multiple grouping expressions, it is important to ensure that they are listed in the appropriate order—if one kind of group is logically contained inside another (for example, departments within a division), that group should be listed after the group that it is contained in (i.e., GROUP BY ROLLUP (DIVISION, DEPARTMENT)), and not before.

The GROUP BY CUBE Clause

The GROUP BY CUBE clause is used to analyze a collection of data by organizing it into groups in multiple dimensions. Thus, if you were to execute a SELECT statement that looks something like this:

```sql
SELECT SEX, WORKDEPT, AVG(SALARY) AS AVG_SALARY
FROM EMPLOYEE
GROUP BY CUBE (SEX, WORKDEPT)
```

you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>SEX</th>
<th>WORKDEPT</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>-</td>
<td>A00</td>
<td>42833.33</td>
</tr>
<tr>
<td>-</td>
<td>B01</td>
<td>41250.00</td>
</tr>
<tr>
<td>-</td>
<td>C01</td>
<td>30156.66</td>
</tr>
<tr>
<td>-</td>
<td>D11</td>
<td>24677.77</td>
</tr>
<tr>
<td>-</td>
<td>D21</td>
<td>25153.33</td>
</tr>
<tr>
<td>-</td>
<td>E01</td>
<td>24056.66</td>
</tr>
<tr>
<td>-</td>
<td>E11</td>
<td>20998.00</td>
</tr>
<tr>
<td>-</td>
<td>E21</td>
<td>23827.50</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>27303.59</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td>28411.53</td>
</tr>
<tr>
<td>M</td>
<td>-</td>
<td>26545.52</td>
</tr>
<tr>
<td>F</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>F</td>
<td>C01</td>
<td>30156.66</td>
</tr>
</tbody>
</table>
This result set contains average salary information for each department found in the table named EMPLOYEE (the lines that contain a null value in the SEX column and a value in the WORKDEPT column of the result data set returned); average salary information for all employees found in the table named EMPLOYEE, regardless of which department they work in (the line that contains a NULL value for both the SEX and the WORKDEPT columns of the result data set returned); average salary information for each sex (the lines that contain a value in the SEX column and a NULL value in the WORKDEPT column of the result data set returned); and average salary information for each sex in each department available (the remaining lines in the result data set returned).

In other words, the data in the result data set produced is grouped:

- By department only
- By sex only
- By sex and department
- As a single group that contains all sexes and all departments.

The term CUBE is intended to suggest that data is being analyzed in more than one dimension. As you can see in the previous example, data analysis was performed in two dimensions, which resulted in four types of groupings. If the SELECT statement:

```
SELECT SEX, WORKDEPT, JOB, AVG(SALARY) AS AVG_SALARY
FROM EMPLOYEE
GROUP BY CUBE (SEX, WORKDEPT, JOB)
```

had been used instead, data analysis would have been performed in three dimensions, and the data would have been broken into eight types of groupings. Thus, the number of types of groups produced by a CUBE operation can be determined by the formula $2^n$ where $n$ is the number of expressions (dimensions) used in the GROUP BY CUBE clause.
The **HAVING** Clause

The *HAVING* clause is used to apply further selection criteria to columns referenced in a *GROUP BY* clause. This clause behaves like the *WHERE* clause, except that it refers to data that has already been grouped by a *GROUP BY* clause. (The *HAVING* clause is used to tell the DB2 Database Manager how to select the rows to be returned in a result data set from rows that have already been grouped.) Like the *WHERE* clause, the *HAVING* clause is followed by a search condition that acts as a simple test that, when applied to a row of data, will evaluate to “TRUE”, “FALSE”, or “Unknown”. If this test evaluates to “TRUE”, the row is to be returned in the result data set produced; if the test evaluates to “FALSE” or “Unknown”, the row is skipped. In addition, the search condition of a *HAVING* clause can consist of the same predicates that are recognized by the *WHERE* clause.

Thus, if you wanted to obtain the average salary for all departments found in the column named DEPTNAME in a table named DEPARTMENT using salary information stored in a table named EMPLOYEE and you wanted to organize the data retrieved by department, but you are only interested in departments whose average salary is greater than $30,000.00, you could do so by executing a *SELECT* statement that looks something like this:

```
SELECT DEPTNAME, AVG(SALARY) AS AVG_SALARY
FROM DEPARTMENT D, EMPLOYEE E
WHERE E.WORKDEPT = D.DEPTNO
GROUP BY DEPTNAME
HAVING AVG(SALARY) > 30000.00
```

When this statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>DEPTNAME</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFORMATION CENTER</td>
<td>30156.66</td>
</tr>
<tr>
<td>PLANNING</td>
<td>41250.00</td>
</tr>
<tr>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>42833.33</td>
</tr>
<tr>
<td>SUPPORT SERVICES</td>
<td>40175.00</td>
</tr>
</tbody>
</table>

4 record(s) selected.

In this example, each row in the result data set produced contains the department name for every department whose average salary for individuals working in that department is greater than $30,000.00, along with the actual average salary for each department.
The ORDER BY Clause

The ORDER BY clause is used to tell the DB2 Database Manager how to sort and order the rows that are to be returned in a result data set produced in response to a query. When specified, the ORDER BY clause is followed by the name of the column(s) whose data values are to be sorted. Multiple columns can be used for sorting, and each column used can be ordered in either ascending or descending order. If the keyword ASC follows the column's name, ascending order is used, and if the keyword DESC follows the column name, descending order is used. (If neither keyword is used, ascending order is used by default.) Furthermore, when more than one column is identified in an ORDER BY clause, the corresponding result data set is sorted by the first column specified (the primary sort key), then the sorted data is sorted again by the next column specified, and so on until the data has been sorted by each column identified.

So, if you wanted to retrieve values for the columns named LASTNAME, FIRSTNME, and EMPNO in a table named EMPLOYEE in which the value for the EMPNO column is greater than '000200' and you wanted the information sorted by LASTNAME followed by FIRSTNME, you could do so by executing a SELECT statement that looks something like this:

```
SELECT LASTNAME, FIRSTNME, EMPNO
FROM EMPLOYEE
WHERE EMPNO > '000200'
ORDER BY LASTNAME ASC, FIRSTNME ASC
```

When this statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>FIRSTNME</th>
<th>EMPNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOUNOT</td>
<td>JASON</td>
<td>000340</td>
</tr>
<tr>
<td>JEFFERSON</td>
<td>JAMES</td>
<td>000230</td>
</tr>
<tr>
<td>JOHNSON</td>
<td>SYBIL</td>
<td>000260</td>
</tr>
<tr>
<td>JONES</td>
<td>WILLIAM</td>
<td>000210</td>
</tr>
<tr>
<td>LEE</td>
<td>WING</td>
<td>000330</td>
</tr>
<tr>
<td>LUTZ</td>
<td>JENNIFER</td>
<td>000220</td>
</tr>
<tr>
<td>MARINO</td>
<td>SALVATORE</td>
<td>000240</td>
</tr>
<tr>
<td>MEHTA</td>
<td>RAMLAL</td>
<td>000320</td>
</tr>
<tr>
<td>PARKER</td>
<td>JOHN</td>
<td>000290</td>
</tr>
<tr>
<td>PEREZ</td>
<td>MARIA</td>
<td>000270</td>
</tr>
<tr>
<td>SCHNEIDER</td>
<td>ETHEL</td>
<td>000280</td>
</tr>
<tr>
<td>SETRIGHT</td>
<td>MAUDE</td>
<td>000310</td>
</tr>
<tr>
<td>SMITH</td>
<td>DANIEL</td>
<td>000250</td>
</tr>
<tr>
<td>SMITH</td>
<td>PHILIP</td>
<td>000300</td>
</tr>
</tbody>
</table>

14 record(s) selected.
As you can see, the data returned is ordered by employee last names and employee first names. (The LASTNAME values are placed in ascending alphabetical order, and the FIRSTNAME values are also placed in ascending alphabetical order.)

Using the ORDER BY clause is easy if the result data set is composed entirely of named columns. But what happens if the result data set produced needs to be ordered by a summary column or a result column that cannot be specified by name? Because these types of situations can exist, an integer value that corresponds to a particular column’s number can be used in place of the column name with the ORDER BY clause. When integer values are used, the first or leftmost column in the result data set produced is treated as column 1, the next is column 2, and so on. Therefore, you could have produced the same result data set produced earlier by executing a SELECT statement that looks like this:

```
SELECT LASTNAME, FIRSTNAME, EMPNO
FROM EMPLOYEE
WHERE EMPNO > '000200'
ORDER BY 1 ASC, 2 ASC
```

It is important to note that even though integer values are primarily used in the ORDER BY clause to specify columns that cannot be specified by name, they can be used in place of any column name as well.

### The Fetch First Clause

The FETCH FIRST clause is used to limit the number of rows returned to the result data set produced in response to a query. When used, the FETCH FIRST clause is followed by a positive integer value and the keywords ROWS ONLY (or ROW ONLY). This tells the DB2 Database Manager that the user/application executing the query does not want to see more than \( n \) number of rows, regardless of how many rows might exist in the result data set that would be produced were the FETCH FIRST clause not specified.

Thus, if you wanted to retrieve the first 10 values found for the columns named WORKDEPT and JOB from a table named EMPLOYEE, you could do so by executing a SELECT statement that looks something like this:

```
SELECT WORKDEPT, JOB FROM EMPLOYEE
FETCH FIRST 10 ROWS ONLY
```

When this SELECT statement is executed, you might see a result data set that looks something like this:
Joining Tables

Most of the examples we have looked at so far have involved only one table. However, one of the more powerful features of the SELECT statement (and the element that makes data normalization possible) is the ability to retrieve data from two or more tables by performing what is known as a *join operation*. A join is a binary operation on two (not necessarily distinct) tables that produces a “derived” table as a result. (If you go back through the examples that have been presented so far, you will see an occasional “sneak preview” of a join operation—particularly in the examples provided for the IN predicate and the HAVING and GROUP BY clauses.)

In its simplest form, the syntax for a SELECT statement that performs a join operation is:

```
SELECT * FROM [ TableName | ViewName, ...]
```

where:

- **TableName** Identifies the name(s) assigned to one or more tables that data is to be retrieved from.
- **ViewName** Identifies the name(s) assigned to one or more views that data is to be retrieved from.

Consequently, if you wanted to retrieve all values stored in a base table named DEPARTMENT and all values stored in a base table named ORG, you could do so by executing a SELECT statement that looks something like this:

```
SELECT * FROM DEPARTMENT, ORG
```

When such a statement is executed, the result data set produced will contain all possible combinations of the rows found in each table specified (otherwise
known as the Cartesian product). Every row in the result data set produced is a row from the first referenced table concatenated with a row from the second referenced table, concatenated in turn with a row from the third referenced table, and so on. The total number of rows found in the result data set produced is the product of the number of rows in all the individual tables referenced. Thus, if the table named DEPARTMENT in our previous example contains five rows and the table named ORG contains two rows, the result data set produced by the statement `SELECT * FROM DEPARTMENT, ORG` will consist of 10 rows (2 x 5 = 10).

A Cartesian product join operation should be used with extreme caution when working with large tables; the amount of resources required to perform such a join operation can have a serious negative impact on performance.

A more common join operation involves collecting data from two or more tables that have one specific column in common and combining the results to create an intermediate result table that contains the values needed to resolve a query. The syntax for a `SELECT` statement that performs this type of join operation is:

```
SELECT
   [* | [Expression] <<AS> [NewColumnName]> ,....]
FROM [[TableName] <<AS> [CorrelationName]> ,....]
[JoinCondition]
```

where:

- **Expression** Identifies one or more columns whose values are to be returned when the `SELECT` statement is executed. The value specified for this option can be any valid SQL language element; however, column names that correspond to the table or view specified in the `FROM` clause are commonly used.

- **NewColumnName** Identifies a new column name that is to be used in place of the corresponding table or view column name specified in the result data set returned by the `SELECT` statement.

- **TableName** Identifies the name(s) assigned to one or more tables that data is to be retrieved from.

- **CorrelationName** Identifies a shorthand name that can be used when referencing the table name specified in the `TableName` parameter.
JoinCondition Identifies the condition to be used to join the tables specified. Typically, this is a \texttt{WHERE} clause in which the values of a column in one table are compared with the values of a similar column in another table.

Thus, a simple join operation could be conducted by executing a \texttt{SELECT} statement that looks something like this:

\begin{verbatim}
SELECT LASTNAME, DEPTNAME
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.WORKDEPT = D.DEPTNO
\end{verbatim}

When this \texttt{SELECT} statement is executed, you might see a result data set that looks something like this:

\begin{verbatim}
LASTNAME        DEPTNAME
--------        ----------------------------
HAAS            SPIFY COMPUTER SERVICE DIV.
THOMPSON        PLANNING
KWAN            INFORMATION CENTER
GEYER           SUPPORT SERVICES
STERN           MANUFACTURING SYSTEMS
PULASKI         ADMINISTRATION SYSTEMS
HENDERSON       OPERATIONS
SPENGER         SOFTWARE SUPPORT
LUCCHESSI       SPIFY COMPUTER SERVICE DIV.
O'CONNELL       SPIFY COMPUTER SERVICE DIV.
QUINTANA        INFORMATION CENTER
NICHOLLS        INFORMATION CENTER
ADAMSON         MANUFACTURING SYSTEMS
LUTZ            MANUFACTURING SYSTEMS
SMITH           ADMINISTRATION SYSTEMS
PEREZ           ADMINISTRATION SYSTEMS
SCHNEIDER       OPERATIONS
PARKER          OPERATIONS
SETRIGHT        OPERATIONS
MEHTA           SOFTWARE SUPPORT
GOUNOT          SOFTWARE SUPPORT
\end{verbatim}

21 record(s) selected.

This type of join is referred to as an \textit{inner join}. Aside from a Cartesian product, only two types of joins can exist: inner joins and \textit{outer joins}. As you might imagine, a significant difference exists between the two.
Inner Joins

An inner join can be thought of as the cross product of two tables, in which every row in one table that has a corresponding row in another table is combined with that row to produce a new record. This type of join works well as long as every row in the first table has a corresponding row in the second table. However, if this is not the case, the result table produced may be missing rows found in either or both of the tables that were joined. Earlier, we saw the most common `SELECT` statement syntax used to perform an inner join operation. The following syntax can also be used to create a `SELECT` statement that performs an inner join operation:

```
SELECT [* | [Expression] <AS> [NewColumnName]>, ...]
FROM [[TableName1] <AS> [CorrelationName1]]
<INNER> JOIN
[[TableName2] <AS> [CorrelationName2]]
ON [JoinCondition]
```

where:

- `Expression` Identifies one or more columns whose values are to be returned when the `SELECT` statement is executed. The value specified for this option can be any valid SQL language element; however, column names that correspond to the table or view specified in the `FROM` clause are commonly used.

- `NewColumnName` Identifies a new column name to be used in place of the corresponding table or view column name specified in the result data set returned by the `SELECT` statement.

- `TableName1` Identifies the name assigned to the first table that data is to be retrieved from.

- `CorrelationName1` Identifies a shorthand name that can be used when referencing the leftmost table of the join operation.

- `TableName2` Identifies the name assigned to the second table that data is to be retrieved from.

- `CorrelationName2` Identifies a shorthand name that can be used when referencing the rightmost table of the join operation.

- `JoinCondition` Identifies the condition to be used to join the two specified tables.
Consequently, the same inner join operation we looked at earlier could be conducted by executing a `SELECT` statement that looks something like this:

```sql
SELECT LASTNAME, DEPTNAME
FROM EMPLOYEE E INNER JOIN DEPARTMENT D
ON E.WORKDEPT = D.DEPTNO
```

Figure 3–1 illustrates how such an inner join operation would work.

It is important to note that inner join queries like the one just shown are typically written without using the keywords `INNER JOIN`. Thus, the previous query would be more likely to be coded like this:

```sql
SELECT LASTNAME, DEPTNAME
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.WORKDEPT = D.DEPTNO
```

<table>
<thead>
<tr>
<th>EMPLOYEE TABLE</th>
<th>DEPARTMENT TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNO</td>
<td>LASTNAME</td>
</tr>
<tr>
<td>001</td>
<td>JAGGER</td>
</tr>
<tr>
<td>002</td>
<td>RICHARDS</td>
</tr>
<tr>
<td>003</td>
<td>WOOD</td>
</tr>
<tr>
<td>004</td>
<td>WATTS</td>
</tr>
<tr>
<td>005</td>
<td>WYMAN</td>
</tr>
<tr>
<td>006</td>
<td>JONES</td>
</tr>
</tbody>
</table>

Record for WYMAN is not in the result data set produced because it has no corresponding DEPTNO value; likewise, records for ENGINEERING and SALES are not in the result data set produced because they have no corresponding WORKDEPT value.

Figure 3–1  A simple inner join operation.
Outer Joins

Outer join operations are used when a join operation is needed and when any rows that would normally be eliminated by an inner join operation need to be preserved. With DB2 UDB, three types of outer joins are available:

**Left outer join.** When a left outer join operation is performed, rows that would have been returned by an inner join operation, together with rows stored in the leftmost table of the join operation (i.e., the table listed on the left side of the `OUTER JOIN` operator) that would have been eliminated by the inner join operation, are returned in the result data set produced.

**Right outer join.** When a right outer join operation is performed, rows that would have been returned by an inner join operation, together with rows stored in the rightmost table of the join operation (i.e., the table listed on the right side of the `OUTER JOIN` operator) that would have been eliminated by the inner join operation, are returned in the result data set produced.

**Full outer join.** When a full outer join operation is performed, rows that would have been returned by an inner join operation, together with rows stored in both tables of the join operation that would have been eliminated by the inner join operation, are returned in the result data set produced.

To understand the basic principles behind an outer join operation, it helps to look at an example. Suppose Table A and Table B are joined by an ordinary inner join operation. Any row in either Table A or Table B that does not have a matching row in the other table (according to the rules of the join condition) is eliminated from the final result data set produced. By contrast, if Table A and Table B are joined by an outer join, any row in either Table A or Table B that does not contain a matching row in the other table is included in the result data set (exactly once), and columns in that row that would have contained matching values from the other table are assigned a null value. Thus, an outer join operation adds non-matching rows to the final result data set produced where an inner join operation excludes them. A left outer join of Table A with Table B preserves all nonmatching rows found in Table A, a right outer join of Table A with Table B preserves all nonmatching rows found in Table B, and a full outer join preserves nonmatching rows found in both Table A and Table B.

The basic syntax for a `SELECT` statement used to perform an outer join operation is:

```
SELECT
    [* | [Expression] <<AS> [NewColumnName]>,...]
```
FROM [[TableName1] <<AS> [CorrelationName1]>]
[LEFT | RIGHT | FULL] OUTER JOIN
[[TableName2] <<AS> [CorrelationName2]>
ON [JoinCondition]

where:

*Expression* Identifies one or more columns whose values are to be returned when the `SELECT` statement is executed. The value specified for this option can be any valid SQL language element; however, column names that correspond to the table or view specified in the `FROM` clause are commonly used.

*NewColumnName* Identifies a new column name that is to be used in place of the corresponding table or view column name specified in the result data set returned by the query.

*TableName1* Identifies the name assigned to the first table that data is to be retrieved from. This table is considered the “left” table in an outer join.

*CorrelationName1* Identifies a shorthand name that can be used when referencing the leftmost table of the join operation.

*TableName2* Identifies the name assigned to the second table that data is to be retrieved from. This table is considered the “right” table in an outer join.

*CorrelationName2* Identifies a shorthand name that can be used when referencing the rightmost table of the join operation.

*JoinCondition* Identifies the condition to be used to join the two tables specified.

Thus, a simple left outer join operation could be conducted by executing a `SELECT` statement that looks something like this:

```sql
SELECT LASTNAME, DEPTNAME
FROM EMPLOYEE E LEFT OUTER JOIN DEPARTMENT D
ON E.WORKDEPT = D.DEPTNO
```

The same query could be used to perform a right outer join operation or a full outer join operation by substituting the keyword `RIGHT` or `FULL` for the keyword `LEFT`. Figure 3–2 illustrates how such a left outer join operation would work; Figure 3–3 illustrates how such a right outer join operation would work; and Figure 3–4 illustrates how such a full join operation would work.
Combining Two or More Queries with a Set Operator

With DB2 UDB, it is possible to combine two or more queries into a single query by using a special operator known as a set operator. When a set operator is used, the results of each query executed are combined in a specific manner to produce a single result data set. The following set operators are available:

**UNION.** When the `UNION` set operator is used, the result data sets produced by each individual query are combined, and all duplicate rows are eliminated.

---

### Figure 3–2
A simple left outer join operation.
When the UNION ALL set operator is used, the result data sets produced by each individual query are combined, and any duplicate rows found are retained.

EXCEPT. When the EXCEPT set operator is used, duplicate rows found in each result data set produced are eliminated from the result data set of the first query and this modified result data set is returned.

EXCEPT ALL. When the EXCEPT ALL set operator is used, all rows found in the first result data set produced that do not have a matching row in the second result data set are returned. (Duplicate rows in the first result data set are retained.)

INTERSECT. When the INTERSECT set operator is used, the result data sets produced by each individual query are compared, and every
record that is found in both result data sets is copied to a new result data set, all duplicate rows in this new result data set are eliminated, and the new result data set is returned.

**INTERSECT ALL.** When the INTERSECT ALL set operator is used, the result data sets produced by each individual query are compared, and each record that is found in both result data sets is copied to a new result data set; all duplicate rows found in this new result data set are retained.

For two result data sets to be combined with a set operator, both must have the same number of columns, and each of those columns must have the same data types assigned to them. So when would you want to combine the results of two queries by using a set operator? Suppose your company keeps individual employee expense account information in a table whose contents are archived at
the end of each fiscal year. When a new fiscal year begins, expenditures for that year are essentially recorded in a new table. Now suppose that, for tax purposes, you need a record of all employees’ expenses for the last two years. To obtain this information, each archived table must be queried, and the results must then be combined. Rather than do this by running individual queries against the archived tables and storing the results in some kind of temporary table, you could perform this operation simply by using the **UNION** set operator with two **SELECT** SQL statements. Such a combination might look something like this:

```
SELECT * FROM EMP_EXP_02
UNION
SELECT * FROM EMP_EXP_01
ORDER BY EXPENSES DESC
```

Figure 3–5 illustrates how such a set operation would work.

Because the record for JAGGER is identical in both tables, it occurs only once in the result data set produced; had the **UNION ALL** set operator been used instead of the **UNION** set operator, both records for JAGGER would have appeared in the result data set produced.

---

**Figure 3–5** A simple UNION set operation.
The same set of queries could be combined using the \texttt{UNION ALL}, \texttt{EXCEPT}, \texttt{EXCEPT ALL}, \texttt{INTERSECT}, or \texttt{INTERSECT ALL} set operator simply by substituting the appropriate keywords for the keyword \texttt{UNION}. However, the results of each operation would be significantly different.

\section*{Using SQL Functions to Transform Data}

Along with a rich set of SQL statements, DB2 UDB comes with a set of built-in functions that can return a variety of data values or convert data values from one data type to another. (A function is an operation denoted by a function name followed by a pair of parentheses enclosing zero or more arguments.) Most of the built-in functions provided by DB2 UDB are classified as being \textit{aggregate} (or \textit{column}, because they work on all values of a column), \textit{scalar} (because they work on a single value in a table or view), \textit{row} (because they return a single row to the SQL statement that references them), or \textit{table} (because they return multiple rows to the SQL statement that references them).

The argument of a column function is a collection of like values. A column function returns a single value (possibly null) and can be specified in an SQL statement wherever an expression can be used. Some of the more common column functions include:

\begin{itemize}
\item \texttt{SUM(\textit{Column})}. Returns the sum of the values in the column specified.
\item \texttt{AVG(\textit{Column})}. Returns the sum of the values in the column specified divided by the number of values found in that column (the average).
\item \texttt{MIN(\textit{Column})}. Returns the smallest value found in the column specified.
\item \texttt{MAX(\textit{Column})}. Returns the largest value found in the column specified.
\item \texttt{COUNT(\textit{Column})}. Returns the total number of non-null values found in the column specified.
\end{itemize}

The arguments of a scalar function are individual scalar values, which can be of different data types and can have different meanings. A scalar function returns a single value (possibly null) and can be specified in an SQL statement wherever an expression can be used. Some of the more common scalar functions include:

\begin{itemize}
\item \texttt{ABS(\textit{Value})}. Returns the absolute value of the value specified.
\item \texttt{COALESCE(\textit{Expression}, \textit{Expression},...)}. Returns the first expression found in the list provided that it is not null (for example, \texttt{COALESCE(EMPID, 0)} returns the value for EMPID unless that value is null, in which case the value 0 will be returned instead).
\end{itemize}
LENGTH('CharacterString'). Returns the number of bytes found in the character string value specified.

LCASE('CharacterString') or LOWER('CharacterString'). Returns a character string in which all of the characters in the character string value specified are converted to lowercase characters.

UCASE('CharacterString') or UPPER('CharacterString'). Returns a character string in which all of the characters in the character string value specified are converted to uppercase characters.

DATE(Value | 'CharacterString'). Returns a date value from a numeric value or string.

MONTH('DateValue'). Returns the month portion of the date value specified.

DAY('DateValue'). Returns the day portion of the date value specified.

YEAR('DateValue'). Returns the year portion of the date value specified.

Row and table functions are special functions because they return one or more rows to the SQL statement that reference them and can only be specified in the FROM clause of a SELECT statement. Such functions are typically used to work with data that does not reside in a DB2 UDB database and/or to convert such data into a format that resembles that of a DB2 table. (The built-in function SNAPSH TO _TABLE is an example of a table function.)

A complete listing of the functions available with DB2 UDB can be found in the IBM DB2 Universal Database, Version 8 SQL Reference Volume 1 product documentation.

A Word About User-Defined Functions

User-defined functions (UDFs) are special objects that are used to extend and enhance the support provided by the built-in functions available with DB2 UDB. Like user-defined data types (UDTs), user-defined functions (or methods) are created and named by a database user. A user-defined function can be an external function written in a high-level programming language, an internal function written entirely in SQL, or a sourced function whose implementation is inherited from another function that already exists.

Like built-in functions, user-defined functions are classified as being scalar, column, row, or table in nature. (User-defined functions are often created to provide the same functionality for user-defined data types that built-in functions provide for the built-in data types upon which user-defined data types are based.) User-defined data types and user-defined functions are covered in much more detail in Chapter 8, “User Defined Routines.”
Common Table Expressions

*Common table expressions* are mechanisms that are used to construct local temporary tables that reside in memory and only exist for the life of the SQL statement that defines them. (In fact, the table that is created in response to a common table expression can only be referenced by the SQL statement that created it.) Common table expressions are typically used:

- In place of a view (when the creation of a view is undesirable, when general use of a view is not required, and when positioned update or delete operations are not used)
- To enable grouping by a column that is derived from a subquery or a scalar function that performs some external action
- When the desired result table is based on host variables
- When the same result table needs to be used by several different queries
- When the results of a query need to be derived using recursion

The temporary table associated with a common table expression is created by prefixing a *SELECT* SQL statement with the *WHERE* keyword. The basic syntax for using this keyword is:

```sql
WITH [CommonTableName] <\( [ColumnName] ,... )> AS ([SELECTStatement])
```

where:

- **CommonTableName** Identifies the name to be assigned to the temporary table to be created.
- **ColumnName** Identifies the name(s) of one or more columns that are to be included in the temporary table to be created. If a list of column names is specified, the number of column names provided must match the number of columns that will be returned by the *SELECT* statement used to create the temporary table. (If a list of column names is not provided, the columns of the temporary table will inherit the names that are assigned to the columns returned by the *SELECT* statement used to create the temporary table.
- **SELECTStatement** Identifies a *SELECT* SQL statement that, when executed, will produce data that will populate the temporary table to be created.
Thus, if you wanted to use a common table expression to create a temporary table that contains a list of all employees (along with their department names) whose current salary is greater than or equal to $25,000.00 and if you then wanted to use this table to find out which of these employees are female, you could do so by executing a SELECT SQL statement that looks something like this:

```
WITH SOME_EMPLOYEES AS
  (SELECT LASTNAME, SALARY, SEX, DEPTNAME
   FROM EMPLOYEE E, DEPARTMENT D
   WHERE E.WORKDEPT = D.DEPTNO AND SALARY >= 25000.00)
SELECT * FROM SOME_EMPLOYEES WHERE SEX = 'F'
```

When this statement is executed, you might see a result data set that looks something like this:

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>SALARY</th>
<th>SEX</th>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAAS</td>
<td>52750.00</td>
<td>F</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>KWAN</td>
<td>38250.00</td>
<td>F</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>NICHOLLS</td>
<td>28420.00</td>
<td>F</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>LUTZ</td>
<td>29840.00</td>
<td>F</td>
<td>MANUFACTURING SYSTEMS</td>
</tr>
<tr>
<td>PULASKI</td>
<td>36170.00</td>
<td>F</td>
<td>ADMINISTRATION SYSTEMS</td>
</tr>
<tr>
<td>PEREZ</td>
<td>27380.00</td>
<td>F</td>
<td>ADMINISTRATION SYSTEMS</td>
</tr>
<tr>
<td>HENDERSON</td>
<td>29750.00</td>
<td>F</td>
<td>OPERATIONS</td>
</tr>
<tr>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>F</td>
<td>OPERATIONS</td>
</tr>
</tbody>
</table>

8 record(s) selected.

Multiple common table expressions can be specified following the single WITH keyword, and each common table expression specified can be referenced by name in the FROM clause of subsequent common table expressions. However, if multiple common table expressions are defined within the same WITH keyword, the table name assigned to each temporary table created must be unique from all other table names used in the SELECT statement that creates them. It is also important to note that the table name assigned to the temporary table created by a common table expression will take precedence over any existing table, view, or alias (in the system catalog) that has the same qualified name and that the SELECT SQL statement that references the original table, view, or alias will actually be working with the temporary table created. (Existing tables, views, and aliases whose names match that of the temporary table are not altered but are simply no longer accessible.)
Retrieving Results from a Result Data Set Using a Cursor

So far, we have looked at a variety of ways in which a query can be constructed using the `SELECT` SQL statement. And we have seen how the results of a query in some cases can be returned to the user when an SQL statement is executed from the Command Line Processor. However, we have not seen how the results of a query can be obtained when a `SELECT` statement is executed from an application program.

When a query is executed from within an application, DB2 UDB uses a mechanism known as a cursor to retrieve data values from the result data set produced. The name “cursor” probably originated from the blinking cursor found on early computer screens, and just as that cursor indicated the current position on the screen and identified where typed words would appear next, a DB2 UDB cursor indicates the current position in the result data set (i.e., the current row) and identifies which row of data will be returned to the application next. Depending upon how it has been defined, a cursor can fall into one of three categories:

**Read-only.** Read-only cursors are cursors that have been constructed in such a way that rows in their corresponding result data set can be read but not modified or deleted. A cursor is considered read-only if it is based on a read-only `SELECT` statement. (For example, the statement `SELECT DEPTNAME FROM DEPARTMENT` is a read-only `SELECT` statement.)

**Updatable.** Updatable cursors are cursors that have been constructed in such a way that rows in their corresponding result data set can be modified or deleted. A cursor is considered updatable if the `FOR UPDATE` clause was specified when the cursor was created. (Only one table can be referenced in the `SELECT` statement that is used to create an updatable cursor.)

**Ambiguous.** Ambiguous cursors are cursors that have been constructed in such a way that it is impossible to tell if they are meant to be read-only or updatable. (Ambiguous cursors are treated as read-only cursors if the BLOCKING ALL option was specified during precompiling or binding. Otherwise, they are considered updatable.)

Regardless of which type of cursor is used, the following steps must be followed if a cursor is to be incorporated into an application program:

1. Declare (define) the cursor along with its type and associate it with the desired query (`SELECT` SQL statement).
2. Open the cursor. This action will cause the corresponding query to be executed and a result data set to be produced.
3. Retrieve (fetch) each row in the result data set, one by one, until an “End of data” condition occurs—each time a row is retrieved from the result data set, the cursor is automatically moved to the next row.

4. If appropriate, modify or delete the current row (but only if the cursor is updatable).

5. Close the cursor. This action will cause the result data set that was produced when the corresponding query was executed to be deleted.

With DB2 UDB (as with most other relational database management systems), the following SQL statements are used to carry out the preceding steps:

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

**The DECLARE CURSOR Statement**

Before a cursor can be used in an application program, it must be created and associated with the SELECT statement that will be used to generate its corresponding result data set. This is done by executing the DECLARE CURSOR SQL statement. The basic syntax for this statement is:

```
DECLARE CURSOR [CursorName]
<WITH HOLD>
<WITH RETURN <TO CLIENT | TO CALLER>>
FOR [[SELECTStatement] | [StatementName]]
<FOR READ ONLY | FOR FETCH ONLY |
 FOR UPDATE <OF [ColumnName,...]>]
```

where:

- **CursorName** identifies the name to be assigned to the cursor to be created.
- **SELECTStatement** identifies a SELECT SQL statement that, when executed, will produce a result data set that is to be associated with the cursor to be created.
- **StatementName** identifies a prepared SELECT SQL statement that, when executed, will produce a result data set that is to be associated with the cursor to be created. (This SELECT statement must be prepared with the PREPARE SQL statement...
before it is used to create a cursor; this statement can contain parameter markers.

**ColumnName**

Identifies the name of one or more columns in the result data set to be produced whose values can be modified by performing a positioned update or a positioned delete operation. (Each name provided must identify an existing column in the result data set produced.)

If the **WITH HOLD** option is specified when the **DECLARE CURSOR** statement is executed, the cursor created will remain open (once it has been opened) across transaction boundaries and must be explicitly closed. (If this option is not used, the scope of the cursor is limited to the transaction in which it is defined and will be closed automatically when the transaction that declares and opens the cursor is terminated.) If the **WITH RETURN** option is specified when the **DECLARE CURSOR** statement is executed, it is assumed that the cursor has been created from within a stored procedure and that once opened, the cursor is to remain open when control is passed back to either the calling application or the client application, depending on how the **WITH RETURN** option was specified.

The clauses **FOR READ ONLY**, **FOR FETCH ONLY**, and **FOR UPDATE <OF [ColumnName,...]>** are actually part of the **SELECT** statement used to build the result data set associated with the cursor and are not part of the **DECLARE CURSOR** statement's syntax. As you might imagine, the use (or lack) of these clauses determine whether the cursor to be created will be a read-only, updatable, or ambiguous cursor.

Thus, if you wanted to define a read-only cursor named **MY_CURSOR** that is associated with a result data set that contains values obtained from the columns **WORKDEPT** and **JOB** found in a table named **EMPLOYEE**, you could do so by executing a **DECLARE CURSOR** statement that looks something like this:

```
DECLARE MY_CURSOR CURSOR
FOR SELECT WORKDEPT, JOB FROM EMPLOYEE
FOR READ ONLY
```

Multiple cursors can be created within a single application; however, each cursor created must have a unique name (within the same source code file).

**The OPEN Statement**

Although a cursor is defined when the **DECLARE CURSOR** SQL statement is executed, the result data set associated with the cursor is not actually produced
until the cursor is opened; when a cursor is opened, all rows that satisfy the query associated with the cursor's definition are retrieved and copied to a result data set. Cursors are opened by executing the `OPEN SQL` statement. The basic syntax for this statement is:

```
OPEN [CursorName]
<USING [HostVariable],... |
    USING DESCRIPTOR [DescriptorName]>
```

where:

- **CursorName** Identifies the name to be assigned to the cursor to be opened.
- **HostVariable** Identifies one or more host variables that are to be used to provide values for any parameter markers that were coded in the `SELECT` statement used to create the cursor to be opened. (Host variables and parameter markers are described in detail in Chapter 4, “Embedded SQL Programming.”)
- **DescriptorName** Identifies an SQL Descriptor Area (SQLDA) data structure variable that contains descriptions of each host variable that is to be used to provide values for parameter markers coded in the `SELECT` statement used to create the cursor to be opened. (The SQLDA data structure variable is described in detail in Chapter 4, “Embedded SQL Programming.”)

Thus, if you wanted to open a cursor named MY_CURSOR (which, in turn, would cause the corresponding result data set to be produced), you could do so by executing an `OPEN` statement that looks like this:

```
OPEN MY_CURSOR
```

On the other hand, if you wanted to open a cursor named MY_CURSOR and associate two host variables (named LastName and FirstName) with parameter markers that were coded in the `SELECT` statement that was used to create the cursor to be opened, you could do so by executing an `OPEN` statement that looks like this:

```
OPEN MY_CURSOR USING :LastName, :FirstName
```

It is important to note that the rows of the result data set associated with a query may be derived during the execution of the `OPEN` statement (in which case a temporary table may be created to hold them); or they may be derived
during the execution of each subsequent FETCH statement. In either case, when a cursor is opened, it is placed in the “Open” state, and the cursor pointer is positioned before the first row of data in the result data set produced; if the result data set is empty, the position of the cursor is effectively “after the last row,” and any subsequent FETCH operations performed will generate a NOT FOUND (+100) condition.

It is important to note that once a cursor has been opened, it can be in one of three possible positions: “Before a Row of Data,” “On a Row of Data,” or “After the Last Row of Data.” If a cursor is positioned “Before a Row of Data,” it will be moved just before the first row of the result data set, and the data values stored in that row will be assigned to the appropriate host variables when the FETCH statement is executed. If a cursor is positioned “On a Row of Data” when the FETCH statement is executed, it will be moved to the next row in the result data set (if one exists), and the data values stored in that row will be assigned to the appropriate host variables. If a cursor is positioned on the last row of the result data set when the FETCH statement is executed, it will be moved to the “After the Last Row of Data” position, the value +100 will be assigned to the sqlcode field of the current SQLCA data structure variable, and the value “02000” will be assigned to the sqlstate field of the current SQLCA data structure variable. (In this case, no data is copied to the host variables specified.)

The FETCH Statement

Once a cursor has been opened, data is retrieved from its associated result data set by calling the FETCH statement repeatedly until all records have been processed. The basic syntax for the FETCH statement is:

```
FETCH <FROM> [CursorName] INTO [HostVariable ,...]
```

or

```
FETCH <FROM> [CursorName] USING DESCRIPTOR [DescriptorName]
```

where:

- **CursorName** Identifies the name assigned to the cursor that data is to be retrieved from.
- **HostVariable** Identifies one or more host variables that values obtained from the result data set associated with the cursor specified are to be copied to.
- **DescriptorName** Identifies an SQL Descriptor Area (SQLDA) data structure variable that contains descriptions of each host variable that values obtained from the result data set associated with the cursor specified are to be copied to.
Thus, if you wanted to retrieve a record from the result data set associated
with a cursor named MY_CURSOR and copy the values obtained to two host
variables named DeptNumber and DeptName, you could do so by executing
a FETCH statement that looks something like this:

    FETCH FROM MY_CURSOR CURSOR INTO :DeptNumber, :DeptName

### The CLOSE Statement

When all records stored in the result data set associated with a cursor have
been retrieved (and copied to host variables) or when the result data set asso-
ciated with a cursor is no longer needed, it can be destroyed by executing the
CLOSE SQL statement. The syntax for this statement is:

    CLOSE [CursorName]
    <WITH RELEASE>

where:

    CursorName    Identifies the name assigned to the cursor to be
closed.

If the WITH RELEASE option is specified when the CLOSE statement is executed,
an attempt will be made to release all locks that were acquired on behalf of the
cursor. (It is important to note that not all of the locks acquired are necessar-
ily released; some locks may be held for other operations or activities.)

Therefore, if you wanted to close a cursor named MY_CURSOR and destroy
its associated result data set, you could do so by executing a CLOSE statement
that looks like this:

    CLOSE MY_CURSOR

### Putting It All Together

Now that we have seen how each of the cursor processing statements available
are used, let's examine how they are typically coded in an application. An embed-
ded SQL application written in the C programming language that uses a cursor
to obtain and print employee identification numbers and last names for all
employees who have the job title 'DESIGNER' might look something like this:

    #include <stdio.h>
    #include <stdlib.h>
```c
#include <sql.h>

void main()
{
    /* Include The SQLCA Data Structure Variable */
    EXEC SQL INCLUDE SQLCA;

    /* Declare The SQL Host Memory Variables */
    EXEC SQL BEGIN DECLARE SECTION;
        char      EmployeeNo[7];
        char      LastName[16];
    EXEC SQL END DECLARE SECTION;

    /* Connect To The SAMPLE Database */
    EXEC SQL CONNECT TO SAMPLE USER db2admin USING ibmdb2;

    /* Declare A Cursor */
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT EMPNO, LASTNAME
        FROM EMPLOYEE
        WHERE JOB = 'DESIGNER';

    /* Open The Cursor */
    EXEC SQL OPEN C1;

    /* Fetch The Records */
    while (sqlca.sqlcode == SQL_RC_OK)
    {
        /* Retrieve A Record */
        EXEC SQL FETCH C1
            INTO :EmployeeNo, :LastName

        /* Print The Information Retrieved */
        if (sqlca.sqlcode == SQL_RC_OK)
            printf("%s, %s\n", EmployeeNo, LastName);
    }

    /* Close The Cursor */
    EXEC SQL CLOSE C1;

    /* Issue A COMMIT To Free All Locks */
    EXEC SQL COMMIT;

    /* Disconnect From The SAMPLE Database */
    EXEC SQL DISCONNECT CURRENT;
}
```

Remember, an application can use several cursors concurrently; however, each cursor must have its own unique name and its own set of `DECLARE CURSOR`, `OPEN`, `FETCH`, and `CLOSE` SQL statements.
Retrieving a Single Row of Data

We have just seen how a cursor can be used to process a result data set, and earlier we saw that a result data set can contain any number of rows (including no rows at all). But is a cursor really necessary if it is known in advance that a result data set will only contain one row? The answer to this question is no. If you know in advance that only one row of data will be produced in response to a query, you can copy the contents of that row (record) to host variables within an application program in one of two ways: by executing a special form of the SELECT SQL statement known as the SELECT INTO statement or by executing the VALUES INTO SQL statement.

The SELECT INTO SQL Statement

The SELECT INTO statement is almost identical to the SELECT statement, in both its syntax and its behavior. However, unlike the SELECT statement, the SELECT INTO statement requires a list of valid host variables to be supplied as part of its syntax and can only be used in an embedded SQL application program. Furthermore, it must not return more than one row of data and must be dynamically prepared. The basic syntax for the SELECT INTO SQL statement is:

```
SELECT <DISTINCT>
[* | [Expression] ,...]
FROM [[TableName] | [ViewName]
  <AS> [CorrelationName]>,...]
INTO [HostVariable] ,...]
<WhereClause>
<GroupByClause>
<HavingClause>
<OrderByClause>
<FetchFirstClause>
```

where:

**Expression** Identifies one or more columns for which values are to be returned when the SELECT statement is executed. The value specified for this option can be any valid SQL language element; however, column names that correspond to the table or view specified in the FROM clause are commonly used.

**TableName** Identifies the name(s) assigned to one or more tables that data is to be retrieved from.
ViewName
Identifies the name(s) assigned to one or more views that data is to be retrieved from.

CorrelationName
Identifies a shorthand name that can be used when referencing the table or view that the correlation name is associated with in any of the SELECT statement clauses.

HostVariable
Identifies one or more host variables that values obtained from the result data set produced are to be copied to.

WhereClause
Identifies a WHERE clause that is to be used with the SELECT statement.

GroupByClause
Identifies a GROUP BY clause that is to be used with the SELECT statement.

HavingClause
Identifies a HAVING clause that is to be used with the SELECT statement.

OrderByClause
Identifies an ORDER BY clause that is to be used with the SELECT statement.

FetchFirstClause
Identifies a FETCH FIRST clause that is to be used with the SELECT statement.

When this form of the SELECT statement is executed, all data retrieved is stored in a result data set; if this result data set contains only one record, the first value in that record is copied to the first host variable specified, the second value is copied to the second host variable specified, and so on. On the other hand, if the result data set produced contains more than one record, the operation will fail and an error will be generated. (If the result data set produced is empty, a NOT FOUND warning will be generated.)

Thus, if you wanted to query a database for a single row of data and copy the data for that row into host variables without using a cursor, you could do so by executing a SELECT INTO statement that looks something like this:

SELECT DEPTNO, DEPTNAME FROM DEPARTMENT
INTO :DeptNumber, :DeptName
WHERE DEPTNO = 'B01'

The VALUES INTO SQL Statement

Like the SELECT INTO SQL statement, the VALUES INTO statement can be used to retrieve the data associated with a single record and copy it to one or more host variables. The basic syntax for the VALUES INTO statement is:

VALUES [Expression] INTO [(HostVariable) , ...]
or

VALUES ( [Expression] ,... ) INTO [[HostVariable] ,...]

where:

Expression Identifies one or more values that are to be returned when the VALUES INTO statement is executed. The value specified for this option can be any valid SQL language element; however, DB2 UDB special registers and SQL functions are typically used.

HostVariable Identifies one or more host variables that values obtained from the result data set produced are to be copied to.

As you can see, the VALUES INTO statement cannot be used to construct complex queries in the same way the SELECT INTO statement can. Like the SELECT INTO statement, when the VALUES INTO statement is executed, all data retrieved is stored in a result data set, and if this result data set contains only one record, the first value in that record is copied to the first host variable specified, the second value is copied to the second host variable specified, and so on. On the other hand, if the result data set produced contains more than one record, the operation will fail, and an error will be generated. (If the result data set produced is empty, a NOT FOUND warning will be generated.)

The VALUES INTO statement is often used to obtain the value assigned to one or more of the DB2 UDB special registers available or to obtain the results of one or more SQL functions. (Refer to Chapter 2, “Database Objects and Programming Methods,” for more information about the special registers that are provided with DB2 UDB.) Thus, if you wanted to retrieve the value of the CURRENT PATH special register and copy it to a host variable, you could do so by executing a VALUES INTO statement that looks something like this:

VALUES (CURRENT PATH)
INTO :Path

Transactions

A transaction (also known as a unit of work) is a sequence of one or more SQL operations grouped together as a single unit, usually within an application process. Such a unit is called “atomic” because, like atoms (before fission and fusion were discovered), it is indivisible—either all of its work is carried out,
or none of its work is carried out. A given transaction can perform any number of SQL operations—from a single operation to many hundreds or even thousands, depending on what is considered a “single step” within your business logic. (It is important to note that the longer a transaction is, the more database concurrency decreases and the more resource locks are acquired; this is usually considered the sign of a poorly written application.)

The initiation and termination of a single transaction defines points of data consistency within a database; either the effects of all operations performed within a transaction are applied to the database and made permanent (committed), or the effects of all operations performed are backed out (rolled back) and the database is returned to the state it was in before the transaction was initiated. In either case, all locks that were acquired on behalf of the transaction, with the exception of those locks acquired for held cursors, are immediately released. (However, any data pages that were copied to a buffer pool on behalf of a transaction will remain in the buffer pool until their storage space is needed—at that time, they will be removed.)

In most cases, transactions are initiated the first time an executable SQL statement is executed after a connection to a database has been made or immediately after a preexisting transaction has been terminated. Once initiated, transactions can be implicitly terminated using a feature known as “automatic commit” (in this case, each executable SQL statement is treated as a single transaction, and any changes made by that statement are applied to the database if the statement executes successfully or discarded if the statement fails) or they can be explicitly terminated by executing the COMMIT or the ROLLBACK SQL statement. The basic syntax for these two statements is:

\[
\text{COMMIT <WORK>}
\]

and

\[
\text{ROLLBACK <WORK>}
\]

When the COMMIT statement is used to terminate a transaction, all changes made to the database since the transaction began are made permanent. However, when the ROLLBACK statement is used, all changes made are backed out, and the database is returned to the state it was in just before the transaction began. Figure 3–6 shows the effects of a transaction that was terminated with a COMMIT statement; Figure 3–7 shows the effects of a transaction that was terminated with a ROLLBACK statement.

It is important to remember that commit and rollback operations only have an effect on changes that have been made within the transaction that they terminate. So, to evaluate the effects of a series of transactions, you must be able to identify where each transaction begins, as well as when and how each transac-
tion was terminated. Figure 3–8 shows how the effects of a series of transactions can be evaluated.

Changes made by a transaction that have not been committed are usually inaccessible to other users and applications (unless those users and/or applications are running under the Uncommitted Read isolation level) and can be backed out with a rollback operation. However, once changes made by a transaction have been committed, they become accessible to all other users and/or applications and can only be removed by executing new SQL statements (within a new transaction). What happens if a system failure occurs before a transaction’s changes can be committed? If only the user/application is disconnected (for example, because of a network failure), the DB2 Database Manager backs out all uncommitted changes (by replaying information stored in the transaction log files), and the database is returned to the state it was in just before the transaction that was terminated unexpectedly began. On the other hand, if the
database or the DB2 Database Manager is terminated (for example, because of a hard disk failure or a loss of power), the DB2 Database Manager will try to roll back all open transactions that it finds in the transaction log file the next time the database is restarted (which will take place automatically the next time a user attempts to connect to the database if the database configuration parameter `autorestart` has been set accordingly). Only after this succeeds will the database be placed online again (i.e., made accessible to users and applications).

### Transaction Management and Savepoints

Often, it is desirable to limit the amount of work performed within a single transaction so that locks acquired on behalf of the transaction are released in a timely manner. (When locks are held by one transaction, other transactions
Figure 3–8  Evaluating the effects of a series of transactions.

may be forced to wait for those locks to be freed before they can continue.) Additionally, if a large number of changes are made within a single transaction, it can take a considerable amount of time to back those changes out if the transaction is rolled back. However, using several small transactions to perform a single large task has its drawbacks as well. For one thing, the opportunity for data inconsistency to occur can be increased because business rules may have to cross several transaction boundaries. Furthermore, each time a
commit statement is used to terminate a transaction, the DB2 Database manager must perform extra work to commit the current transaction and start a new one. (Another drawback of having multiple commit points for a particular operation is that portions of an operation might be committed and therefore be visible to other applications before the operation is fully completed.)

To get around these issues, DB2 UDB uses a mechanism known as a savepoint, which allows an application to break the work being performed by a single large transaction into one or more subsets. Using application savepoints avoids the exposure to “dirty data” that might occur when multiple commits are performed while providing granular control over an operation. (You can use as many savepoints as you want within a single transaction; however, savepoints cannot be nested.)

Savepoints are created by executing the SAVEPOINT SQL statement. The basic syntax for this statement is:

```
SAVEPOINT [SavepointName]
<UNIQUE>
ON ROLLBACK RETAIN CURSORS
<ON ROLLBACK RETAIN LOCKS>
```

where:

- **SavepointName** Identifies the name to be assigned to the savepoint to be created.

If the UNIQUE option is specified when the SAVEPOINT statement is executed, the name assigned to the savepoint created will be unique and cannot be reused by the application that created it as long as the savepoint is active.

Thus, if you wanted to create a savepoint named MY_SP, you could do so by executing a SAVEPOINT statement that looks like this:

```
SAVEPOINT MY_SP ON ROLLBACK RETAIN CURSORS
```

Once created, a savepoint can be used in conjunction with a special form of the ROLLBACK SQL statement to return a database to the state it was in at the point in time a particular savepoint was created. The syntax for this form of the ROLLBACK statement is:

```
ROLLBACK <WORK> TO SAVEPOINT <[SavepointName]>
```

where:

- **SavepointName** Identifies the name assigned to the savepoint that all operations performed against the database are to be backed out to.
And finally, when a savepoint is no longer needed, it can be released by executing the `RELEASE SAVEPOINT` SQL statement. The syntax for this statement is:

```
RELEASE <TO> SAVEPOINT <[SavepointName]>
```

where:

- `SavepointName` Identifies the name assigned to the savepoint that is to be released.

The following embedded SQL application (written in the C programming language) illustrates how a savepoint might be used with the `ROLLBACK` SQL statement in a single transaction to control the behavior of the transaction:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>

void main()
{
    /* Include The SQLCA Data Structure Variable */
    EXEC SQL INCLUDE SQLCA;

    /* Connect To The SAMPLE Database */
    EXEC SQL CONNECT TO SAMPLE USER db2admin USING ibmdb2;

    /* Add A Record To The ORDER Table */
    EXEC SQL INSERT INTO ORDER (ITEM) VALUES ('Lamp')

    /* Create And Set The First Savepoint */
    EXEC SQL SAVEPOINT SP1 ON ROLLBACK RETAIN CURSORS;

    /* Add Two More Records To The ORDER Table */
    EXEC SQL INSERT INTO ORDER (ITEM) VALUES ('Radio')
    EXEC SQL INSERT INTO ORDER (ITEM) VALUES ('Power Cord')

    /* Remove The Last Two Records Added To The ORDER Table */
    EXEC SQL ROLLBACK TO SAVEPOINT SP1;

    /* Commit The Transaction (Which Releases The Savepoint) */
    EXEC SQL COMMIT;

    /* Disconnect From The SAMPLE Database */
    EXEC SQL DISCONNECT CURRENT;
}
```
In this example, the last two records added to the ORDERS table are removed when the `ROLLBACK TO SAVEPOINT` SQL statement is executed, but the first record added remains and is externalized to the database when the transaction is committed.

Once a savepoint is created, all subsequent SQL statements executed are associated with that savepoint until it is released either explicitly by calling the `RELEASE SAVEPOINT` statement or implicitly by ending the transaction or unit of work that the savepoint was created in. In addition, when you issue a `ROLLBACK TO SAVEPOINT` SQL statement, the corresponding savepoint is not automatically released as soon as the rollback operation is completed. Instead, you can issue multiple `ROLLBACK TO SAVEPOINT` statements for a given transaction, and each time a `ROLLBACK TO SAVEPOINT` statement is executed, the database will be returned to the state it was in at the time the savepoint was created. (If multiple savepoints have been created, it is possible to rollback to any savepoint available; you are not required to successively rollback to every savepoint, in the opposite order in which they were created, to return the database to the state it was in when an earlier savepoint was created.)
Practice Questions

Question 1

Given the following tables:

<table>
<thead>
<tr>
<th>EMPID</th>
<th>LASTNAME</th>
<th>DEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jagger</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Richards</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Watts</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Wood</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPTID</th>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Planning</td>
</tr>
<tr>
<td>2</td>
<td>Support</td>
</tr>
</tbody>
</table>

Assuming EMPLOYEES.DEPT is a foreign key on DEPARTMENT.DEPTID, if the following statements are executed:

```
INSERT INTO employees VALUES (5, 'Wyman', 2)
INSERT INTO employees VALUES (6, 'Jones', 1)
INSERT INTO employees VALUES (7, 'Stewart', 3)
```

How many rows will be stored in the EMPLOYEES table?

❍ A. 4  
❍ B. 5  
❍ C. 6  
❍ D. 7
Question 2

Given the following tables:

<table>
<thead>
<tr>
<th>COUNTRY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>NAME</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>USA</td>
</tr>
<tr>
<td>2</td>
<td>Canada</td>
</tr>
<tr>
<td>3</td>
<td>Mexico</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NATION</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>NAME</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>USA</td>
</tr>
<tr>
<td>2</td>
<td>Australia</td>
</tr>
<tr>
<td>3</td>
<td>Great Britain</td>
</tr>
</tbody>
</table>

and the code

EXEC SQL DECLARE c1 CURSOR FOR
  SELECT * FROM country
  UNION ALL
  SELECT * FROM nation
EXEC SQL OPEN c1

How many rows will exist in the result data set produced?

- A. 3
- B. 4
- C. 5
- D. 6
Question 3

Given the following tables:

TAB1

<table>
<thead>
<tr>
<th>COL_1</th>
<th>COL_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td>14</td>
</tr>
</tbody>
</table>

TAB2

<table>
<thead>
<tr>
<th>COL_A</th>
<th>COL_B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>21</td>
</tr>
<tr>
<td>C</td>
<td>23</td>
</tr>
<tr>
<td>D</td>
<td>25</td>
</tr>
</tbody>
</table>

and the following query that executes successfully:

```sql
SELECT * FROM tab1 LEFT OUTER JOIN tab2 ON tab1.col_1 = tab2.col_a
```

How many rows will be returned?

- A. 4
- B. 3
- C. 2
- D. 1

Question 4

The following CREATE TABLE statement was used to create a table named TABLEA:

```sql
CREATE TABLE tablea (col1 INTEGER)
```

Which of the following statements can be used to remove all records stored in table TABLEA?

- A. DELETE FROM tablea
- B. DROP * FROM tablea
- C. REMOVE ALL FROM tablea
- D. UPDATE tablea SET col1 = NULL
Question 5

Given the following tables:

<table>
<thead>
<tr>
<th>TAB1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>COL1</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAB2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>COL1</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

If the following SQL statement executes successfully:

```
UPDATE tab1 SET col1 = col1 + 10 WHERE col1 IN (SELECT * FROM tab2)
```

How many rows in TAB1 will be modified?

- A. 0
- B. 1
- C. 2
- D. 3

Question 6

Which of the following will extend the functionality provided by the MIN() built-in function to a distinct data type?

- A. A distinct type extender
- B. A user-defined data type
- C. A sourced user-defined function
- D. A function extender
Question 7

Given the following SQL statements:

```
CREATE TABLE empinfo (name CHAR(10), salary DEC NOT NULL WITH DEFAULT)
INSERT INTO empinfo VALUES ('Smith', 20000)
INSERT INTO empinfo (name) VALUES ('Jones')
INSERT INTO empinfo VALUES ('Doe', 25000)
INSERT INTO empinfo (name) VALUES (NULL)
```

Which two of the following statements will only retrieve one row?

- A. SELECT salary / (SELECT SUM(salary) FROM empinfo) FROM empinfo
- B. SELECT SUM(salary) / COUNT(*) FROM empinfo
- C. SELECT COALESCE(LCASE(name), 'Unknown') FROM empinfo
- D. SELECT salary FROM empinfo WHERE salary IN (SELECT salary / (SELECT SUM(salary) FROM empinfo) FROM empinfo)
- E. SELECT COALESCE(MIN(salary), 0) FROM empinfo

Question 8

Which of the following will return values that are only in upper case?

- A. SELECT name FROM employee WHERE UCASE(name) = 'smith'
- B. SELECT name FROM employee WHERE UCASE(name) = 'SMITH'
- C. SELECT UCASE(name) FROM employee WHERE name = 'smith'
- D. SELECT name FROM employee WHERE name IN (SELECT name FROM employee WHERE UCASE(name) = UCASE('smith'))

Question 9

Given the following SQL statement:

```
WITH emp AS (SELECT lastname, salary FROM employees)
SELECT * FROM emp WHERE salary >= 35000.00
```

What does EMP refer to?

- A. A user table
- B. A declared temporary table
- C. A system catalog table
- D. A local temporary that resides in memory
Question 10

Which of the following is NOT used to retrieve a single row of data?

- A. SELECT
- B. SELECT INTO
- C. DECLARE CURSOR
- D. VALUES

Question 11

Given the following table and the statements below:

<table>
<thead>
<tr>
<th>COL_1</th>
<th>COL_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
</tr>
<tr>
<td>E</td>
<td>50</td>
</tr>
</tbody>
</table>

DECLARE c1 CURSOR WITH HOLD FOR SELECT * FROM tab1 ORDER BY col_1
OPEN c1
FETCH c1
FETCH c1
FETCH c1
FETCH c1
COMMIT
FETCH c1
CLOSE c1
FETCH c1

Which of the following is the last value obtained for COL_2?

- A. 20
- B. 30
- C. 40
- D. 50
Question 12

Which of the following is NOT a characteristic of all cursors used in an embedded SQL application?

- A. Must be declared before they can be used
- B. Must be reserved in the database
- C. Must be unique within a source code file
- D. May be updatable

Question 13

Given the following two tables:

<table>
<thead>
<tr>
<th>TAB1</th>
<th>COL_1</th>
<th>COL_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAB2</th>
<th>COL_A</th>
<th>COL_B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>25</td>
</tr>
</tbody>
</table>

Assuming the following results are desired:

<table>
<thead>
<tr>
<th>COL_1</th>
<th>COL_2</th>
<th>COL_A</th>
<th>COL_B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>A</td>
<td>21</td>
</tr>
<tr>
<td>B</td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>14</td>
<td>C</td>
<td>23</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>D</td>
<td>25</td>
</tr>
</tbody>
</table>

Which of the following joins will produce the desired results?

- A. SELECT * FROM tab1 INNER JOIN tab2 ON col_1 = col_a
- B. SELECT * FROM tab1 LEFT OUTER JOIN tab2 ON col_1 = col_a
- C. SELECT * FROM tab1 RIGHT OUTER JOIN tab2 ON col_1 = col_a
- D. SELECT * FROM tab1 FULL OUTER JOIN tab2 ON col_1 = col_a
Given the following set of statements:

CREATE TABLE tab1 (col1 INTEGER, col2 CHAR(20))
COMMIT
INSERT INTO tab1 VALUES (123, ‘Red’)
INSERT INTO tab1 VALUES (456, ‘Yellow’)
COMMIT
DELETE FROM tab1 WHERE col1 = 123
COMMIT
INSERT INTO tab1 VALUES (789, ‘Blue’)
ROLLBACK
INSERT INTO tab1 VALUES (789, ‘Green’)
ROLLBACK
UPDATE tab1 SET col2 = NULL
COMMIT

Which of the following records would be returned by the statement

```
SELECT * FROM tab1 ?
```

- A. COL1 COL2
  ------- -------
  123  Red
  1 record(s) selected.
- B. COL1 COL2
  ------- -------
  456  Yellow
  1 record(s) selected.
- C. COL1 COL2
  ------- -------
  456  -
  1 record(s) selected.
- D. COL1 COL2
  ------- -------
  789  Green
  1 record(s) selected.
Question 15

Which of the following is NOT a benefit of a user-defined function?

❍ A. Simplifies application maintenance
❍ B. Improves application concurrency
❍ C. Provides built-in function support for user-defined data types
❍ D. Name and calling convention controllable

Question 16

Given the following tables:

<table>
<thead>
<tr>
<th>TABLEA</th>
<th>TABLEB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPID</td>
<td>EMPID</td>
</tr>
<tr>
<td>NAME</td>
<td>WEEKNO</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>USER1</td>
<td>1</td>
</tr>
<tr>
<td>USER2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

and the fact that TABLEB was defined as follows:

```
CREATE TABLE tableb (  empid       SMALLINT,
                        weekno    SMALLINT,
                        payamt     DECIMAL(6,2),
                        CONSTRAINT const1 FOREIGN KEY (empid) REFERENCES tablea(empid)
                        ON DELETE NO ACTION)
```

If the following command is issued:

```
DELETE FROM tablea WHERE empid=2
```

How many rows will be deleted from TABLEA and TABLEB?

❍ A. 0, 0
❍ B. 0, 1
❍ C. 1, 0
❍ D. 1, 1
Question 17

Given the following table:

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>1</td>
</tr>
<tr>
<td>bcde</td>
<td>2</td>
</tr>
<tr>
<td>cdefg</td>
<td>4</td>
</tr>
</tbody>
</table>

Which of the following statements will retrieve the largest computed value?

- A. SELECT SUM(col2) / COUNT(*) FROM tab1
- B. SELECT MAX(col2) FROM tab1
- C. SELECT LENGTH(col1) FROM tab1 WHERE col2 = 4
- D. SELECT STRLEN(col1) FROM tab1 WHERE col2 = 4

Question 18

Which of the following types of functions requires a collection of like values as its input?

- A. SCALAR
- B. COLUMN
- C. TABLE
- D. GROUPING

Question 19

If the following statements are executed:

```
EXEC SQL DECLARE c1 CURSOR FOR
    SELECT deptid, deptname FROM department
EXEC SQL OPEN c1
```

and an empty result data set is produced, what will happen when the following statement is executed?

```
EXEC SQL FETCH c1 into :id, :name
```

- A. The cursor will remain open and the return code -100 will be returned
- B. The cursor will be closed and the return code -100 will be returned
- C. The cursor will remain open and the return code 100 will be returned
- D. The cursor will be closed and the return code 100 will be returned
Question 20

If the following SQL statements are executed:

```
CREATE TABLE tab1 (col1 INT, col2 INT)
COMMIT
INSERT INTO tab1 VALUES (1, 1)
INSERT INTO tab1 VALUES (2, 2)
SAVEPOINT sp1
UPDATE  tab1 SET col1 = 10 WHERE col2 = 1
DELETE FROM  tab1 WHERE col2 = 2
SAVEPOINT sp2
INSERT INTO tab1 VALUES (20, 20)
INSERT INTO tab1 VALUES (40, 40)
ROLLBACK TO SAVEPOINT sp1
INSERT INTO tab1 VALUES (20, 20)
COMMIT
```

Assuming auto commit was not used, how many rows will be returned when the following statement is executed?

```
SELECT * FROM tab1
```

- A. 0
- B. 1
- C. 2
- D. 3
Question 21

Given the following table:

<table>
<thead>
<tr>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>30</td>
</tr>
</tbody>
</table>

Which of the following cursor definitions will create a cursor named CUR1 that can be used to update records found in table T1 without restrictions?

- A. DECLARE cur1 CURSOR FOR SELECT * FROM t1 FOR UPDATE
- B. DECLARE cur1 CURSOR FOR SELECT * FROM t1 FOR UPDATE OF t1
- C. DECLARE cur1 CURSOR FOR UPDATE OF c1 FROM t1
- D. DECLARE cur1 CURSOR FOR SELECT * FROM t1 FOR UPDATE OF c1

Question 22

Which of the following does NOT take place when a transaction is committed?

- A. All cursors opened within the transaction that were not defined with the WITH HOLD clause are destroyed
- B. Locks acquired by the transaction are released
- C. Any data pages that were copied to a buffer pool on behalf of the transaction are deleted
- D. Changes made to data by the transaction are permanently recorded in the database
Answers

Question 1

The correct answer is C. The EMPLOYEES table contained 4 records initially. Then, an attempt was made to add three more records; however, the statement “INSERT INTO employees VALUES (7, 'Stewart', 3)” failed because it violates the insert rule of the referential constraint between the EMPLOYEES table and the DEPARTMENT table (there is no record in the DEPARTMENT table that has a DEPTID value of 3). Thus, only 2 records were added to the EMPLOYEES table, bringing the total number of records to 6.

Question 2

The correct answer is D. When the UNION ALL set operator is used, the result data sets produced by each individual query are combined, and any duplicate rows found are retained. Thus, the result data set produced for cursor C1 will contain all records found in each table.

Question 3

The correct answer is B. When a left outer join operation is performed, rows that would have been returned by an inner join operation, together with all rows stored in the leftmost table of the join operation (i.e., the table listed first in the OUTER JOIN clause) that would have been eliminated by the inner join operation, are returned in the result data set produced. Therefore, the result data set produced by this join operation would look like this:

<table>
<thead>
<tr>
<th>COL_1</th>
<th>COL_2</th>
<th>COL_A</th>
<th>COL_B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>A</td>
<td>21</td>
</tr>
<tr>
<td>B</td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>14</td>
<td>C</td>
<td>23</td>
</tr>
</tbody>
</table>
Question 4

The correct answer is A. Although the UPDATE statement can be used to delete individual values from a base table (by setting those values to NULL), it cannot be used to remove entire rows. When one or more rows of data need to be removed from a base table, the DELETE SQL statement must be used instead. (The DROP statement is used to delete both an object and its data—in this case, the DROP statement is coded incorrectly. There is no REMOVE statement.)

Question 5

The correct answer is D. The IN predicate is used to define a comparison relationship in which a value is checked to see whether it matches a value in a finite set of values. This finite set of values can consist of one or more literal values that are coded directly in the UPDATE statement or it can be composed of the non-null values found in the result data set generated by a subquery. In this case, an update operation is performed on every row in TAB1 that has a matching value in TAB2.

Question 6

The correct answer is C. User-defined functions are often created to provide the same functionality for user-defined data types that built-in functions provide for the built-in data types upon which user-defined data types are based. (A user-defined function can be an external function written in a high-level programming language or a sourced function whose implementation is inherited from some other function that already exists.)

Question 7

The correct answers are B and E. When only columnar functions are used in a SELECT statement, often one row of data is returned. Just the opposite is true when scalar functions are used. The statement "SELECT SUM(salary) / COUNT(*) FROM empinfo" contains only columnar functions, as does the statement "SELECT COALESCE(MIN(salary), 0) FROM empinfo"—in this case, the
COALESCE() function uses output from the MIN() function as its input, so, for all intents and purposes, only one columnar function was used.

Question 8

The correct answer is C. The key wording here is return values in upper case. In order to return values in upper case, the UCASE() function must be used to convert all values retrieved. If the UCASE() function is used in the WHERE clause of a SELECT statement, the conversion is applied for the purpose of locating matching values—not to return values that have been converted to upper case.

Question 9

The correct answer is D. The SQL statement “WITH emp AS (SELECT lastname, salary FROM employees) SELECT * FROM emp WHERE salary >= 35000.00” is a common table expression, and EMP refers to a local temporary table that resides in memory. (Common table expressions are mechanisms used to construct local temporary tables that reside in memory and exist only for the life of the SQL statement that defines them; additionally, tables that are created by a common table expression can be referenced only by the SQL statement that created them.)

Question 10

The correct answer is C. A SELECT statement can be constructed so it only returns one row of data; the SELECT INTO statement must return no more than one row of data, or it will fail—the same is true for the VALUES and VALUES INTO statements. However, a cursor is always used to retrieve multiple rows of data.

Question 11

The correct answer is C. When a cursor that has been declared with the WITH HOLD option specified (as in the example shown) is opened, it will remain open across transaction boundaries until it is explicitly closed; otherwise, it will be
implicitly closed when the transaction that opens it is terminated. In this example, the cursor is opened, the first three rows are fetched from it, the transaction is committed (but the cursor is not closed), another row is fetched from the cursor, and then the cursor is closed. Thus, the last value obtained will be:

<table>
<thead>
<tr>
<th>COL_1</th>
<th>COL_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>40</td>
</tr>
</tbody>
</table>

Question 12

The correct answer is B. All cursors must be declared before they can be used; their names must be unique within the source code file that uses them; and they can be read-only, updatable, or ambiguous. However, because cursors are only used by applications (to retrieve and process multiple rows of data), their definitions are not stored in the database, nor is any space reserved in the database for them.

Question 13

The correct answer is D. When a full outer join operation is performed, rows that would have been returned by an inner join operation and all rows stored in both tables of the join operation that would have been eliminated by the inner join operation are returned in the result data set produced.

Question 14

The correct answer is C. Table TAB1 is created, two rows are inserted, and the first row is deleted because a COMMIT statement follows each of these operations. The next two rows that are inserted are removed because a ROLLBACK statement follows their insertion. And finally, the value for COL2 of all rows is set to null—again because this operation is followed by a COMMIT statement.
Question 15

The correct answer is B. User-defined functions simplify application maintenance (because the processing they perform is managed by the database and not the application that uses them). They are often used to provide the same functionality for user-defined data types that built-in functions provide for the built-in data types that user-defined data types are based on, and their name and calling convention is controlled by the user who creates them. They have no impact on concurrency.

Question 16

The correct answer is A. The ON DELETE NO ACTION definition ensures that when a delete operation is performed on the parent table in a referential constraint, the value for the foreign key of each row in the child table will have a matching value in the parent key of the parent table (after all other referential constraints have been applied). Therefore, no row will be deleted from TABLEA because a row exists in TABLEB that references the row that the DELETE statement is trying to remove. Because the ON DELETE CASCADE definition was not used, no row will be deleted from TABLEB.

Question 17

The correct answer is C. The statement “SELECT SUM(col2) / COUNT(*) FROM tab1” will return the value 2; the statement “SELECT MAX(col2) FROM tab1” will return the value 4; and the statement “SELECT STRLEN(col1) FROM tab1 WHERE col2 = 4” is invalid because STRLEN() is not a valid function. However, the statement “SELECT LENGTH(col1) FROM tab1 WHERE col2 = 4” will return the value 5, which is the largest computed value.

Question 18

The correct answer is B. The argument of a columnar function is a collection of like values. The arguments of a scalar function are individual scalar values, which can be of different data types and can have different meanings. Table functions are special functions because they return a table to the SQL statement that references them and can only be specified in the FROM clause of a SELECT statement. There are no grouping functions.
Question 19

The correct answer is C. When a cursor is opened, it is placed in the “Open” state, and the cursor pointer is positioned before the first row of data in the result data set produced; if the result data set is empty, the position of the cursor is effectively “after the last row,” and any subsequent FETCH operations performed will generate a NOT FOUND (SQLCODE +100, SQLSTATE 02000) condition.

Question 20

The correct answer is D. In this example, the effects of the update and delete operations and the results of the two insert operations that follow the delete operation are removed when the ROLLBACK TO SAVEPOINT sp1 SQL statement is executed. The effects of the first two insert operations and the last insert operation are externalized to the database when the transaction is committed; thus, 3 rows are added to the table TAB1.

Question 21

The correct answer is A. The statement “DECLARE cur1 CURSOR FOR SELECT * FROM t1 FOR UPDATE OF t1” is not valid. Neither is the statement “DECLARE cur1 CURSOR FOR UPDATE OF c1 FROM t1”. The statement “DECLARE cur1 CURSOR FOR SELECT * FROM t1 FOR UPDATE OF c1” creates a cursor that allows only update operations to be performed on column C1. Therefore, the correct answer is A.

Question 22

The correct answer is C. When the COMMIT statement is used to terminate a transaction, all changes made to the database since the transaction began are made permanent, and any locks that were acquired on behalf of the transaction, with the exception of those locks acquired for held cursors, are immediately released. However, any data pages that were copied to a buffer pool on behalf of a transaction will remain in the buffer pool until their storage space is needed—at that time they will be removed. Furthermore, if a cursor is created within a transaction and the WITH HOLD option is specified when the DECLARE CURSOR statement is executed, the cursor created will remain open (once it has been opened) across transaction boundaries and must be explicitly closed.