IBM DB2 Universal Database

SQL Getting Started

Version 6

SC09-2856-00

IBM DB2 Universal Database

SQL Getting Started

Version 6

SC09-2856-00

Before using this information and the product it supports, be sure to read the general information under “Appendix B. Notices” on page 91.

Contents

Welcome

This book is intended for novice users of Structured Query Language (SQL) and relational databases. It will:

- Discuss basic concepts of DB2 SQL.
- Explain how to perform database manipulation tasks.
- Demonstrate tasks through examples.

Before you try out any of the examples in this book, if you are the administrator you should:

- Install and configure the server as outlined in the *Quick Beginnings* book for your operating system. It is recommended that you do not put your own data into the DB2 SAMPLE database.
- Create the DB2 administrator username following the instructions in the *Quick Beginnings* book.

Otherwise, ensure that you have a valid user ID or username and the appropriate authority and privileges.

This book's focus is on providing a solid understanding of DB2 SQL.

Related Documentation for This Book

You may find the following publications useful:

Quick Beginnings	Contains information required to install and use the database manager.
SQL Reference	Contains SQL reference information.
Administration Guide	Contains information required to design, implement, and maintain a database to be accessed either locally or in a client/server environment.
Application Development Guide	Discusses the application development process and how to code, compile, and execute application programs that use embedded SQL to access the database.

Highlighting Conventions

The following conventions are used in this book.

Bold	In examples, it indicates commands and keywords predefined by the system.

Italics	Indicates one of the following: • The introduction of a new term • A reference to another source of information.
UPPERCASE	Indicates one of the following: • Commands and keywords predefined by the system • Examples of specific data values or column names.

Chapter 1. Relational Databases and SQL

In a *relational database*, data is stored in *tables*. A table is a collection of *rows* and *columns*. *Structured Query Language(SQL)* is used to retrieve or update data by specifying columns, tables and various relationships between them.

SQL is a standardized language for defining and manipulating data in a relational database. SQL statements are executed by a *database manager*. A database manager is a computer program that manages the data.

A *partitioned* relational database is a relational database where the data is managed across multiple partitions (also called *nodes*). In this book we will focus our attention on single partition databases.

You can access the sample database and try out all the examples in this book through *interactive* SQL by using an interface like the command line processor (CLP) or the command centre.

Chapter 2. Organizing Data

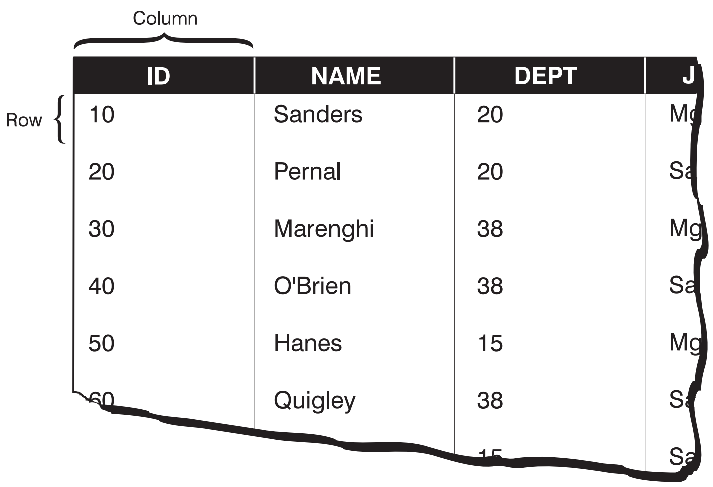
This chapter presents important conceptual descriptions of *tables*, *views* and *schemas*. It's a high level overview showing the connection between different building blocks of a relational database. The last section provides a brief discussion of some of the important and more commonly used data types.

Tables

Tables are logical structures made up of a defined number of columns and a variable number of rows. A *column* is a set of values of the same data type. The rows are not necessarily ordered within a table. To order the result set, you have to explicitly specify ordering in the SQL statement which selects data from the table. At the intersection of every column and row is a specific data item called a *value*. In Figure 1, 'Sanders' is an example of a value in the table.

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

Figure 1 illustrates a section of a table. Columns and rows have been marked.



ID	NAME	DEPT	J
10	Sanders	20	Mg
20	Pernal	20	Sa
30	Marenghi	38	Mg
40	O'Brien	38	Sa
50	Hanes	15	Mg
60	Quigley	38	Sa
		15	Sa

Figure 1. Visualization of a Table

Views

A *view* provides an alternate way of looking at the data in one or more tables. It is a dynamic window on tables.

Views allow multiple users to see different presentations of the same data. For example, several users may be accessing a table of data about employees. One user may see data about some employees but not others, and another may see some data about all employees but not their salaries. Each of these users is operating on a view that is derived from the real table. Each view appears to be a table and has a name of its own.

An advantage of using views is that you can use them to control access to sensitive data. So, different people can have access to different columns or rows of the data.

Schemas

A *schema* is a collection of named objects and provides a logical classification of objects in the database. A schema may contain database objects such as tables and views.

A schema itself is also considered to be an object in the database. It is created implicitly when you create a table or a view. Or, you can create it explicitly using the CREATE SCHEMA statement.

When you create an object, you can *qualify* its name with the name of the particular schema. Named objects have two-part names, where the first part of the name is the name of the schema to which the object is assigned. If you do not specify a schema name, the object is assigned to the default schema whose name is the *authorization ID* of the user executing the statement. For interactive SQL, the method used to execute the examples in this book, authorization ID is the userid specified with the CONNECT statement. For example, if the name of the table is STAFF, and the userid specified in the CONNECT statement is USERID, then the qualified name is USERID.STAFF. See “Connecting to a Database” on page 17 for details on the CONNECT statement.

Some schema names are reserved. For example, *built-in functions* are in the SYSIBM schema while the preinstalled *user-defined functions* belong to the SYSFUN schema. Refer to the *SQL Reference* for details on the CREATE SCHEMA statement.

Data Types

Data types define acceptable values for constants, columns, host variables, functions, expressions and special registers. This section describes the data types referred to in the examples. For a full list and complete description of other data types refer to the *SQL Reference*.

Character String

A *character string* is a sequence of bytes. The length of the string is the number of bytes in the sequence. If the length is zero, the value is called the *empty string*.

Fixed-Length Character String

CHAR(x) is a fixed length string. The length attribute x must be between 1 and 254, inclusive.

Varying-Length Character String

Varying-length character strings are of three types: VARCHAR, LONG VARCHAR, and CLOB. VARCHAR(x) types are varying-length strings, so a string of length 9 can be inserted into VARCHAR(15) but will still have a string length of 9. See “Large Objects (LOBs)” on page 67 for details on CLOB.

Graphic String

A *graphic string* is a sequence of double-byte character data.

Fixed-Length Graphic String

GRAPHIC(x) is a fixed length string. The length attribute x must be between 1 and 127, inclusive.

Varying-Length Graphic String

Varying-length graphic strings are of three types: VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB. See “Large Objects (LOBs)” on page 67 for details on DBCLOB.

Binary String

A *binary string* is a sequence of bytes. It is used to hold nontraditional data such as pictures. Binary Large OBject (BLOB) is a binary string. See “Large Objects (LOBs)” on page 67 for more information.

Numbers

All numbers have a sign and a precision. The precision is the number of bits or digits excluding the sign.

SMALLINT
A *SMALLINT (small integer)* is a two byte integer with a precision of 5 digits.

INTEGER
An *INTEGER (large integer)* is a four byte integer with a precision of 10 digits.

REAL A *REAL (single-precision floating-point number)* is a 32 bit approximation of a real number.

DOUBLE
A *DOUBLE (double-precision floating-point number)* is a 64 bit approximation of a real number. DOUBLE is also referred to as FLOAT.

DECIMAL(p,s)

A *DECIMAL* is a decimal number. The position of the decimal point is determined by the *precision (p)* and the *scale (s)* of the number. Precision is the total number of digits and has to be less than 32. Scale is the number of digits in the fractional part and is always smaller than or equal to the value of precision. The decimal value defaults to precision of 5 and scale of 0 if precision and scale are not specified.

Datetime Values

Datetime values are representations of dates, times, and timestamps. Datetime values can be used in certain arithmetic and string operations and are compatible with certain strings, however they are neither strings nor numbers.[1]

Date A *date* is a three-part value (year, month, and day).

Time A *time* is a three-part value (hour, minute, and second) designating a time of day using a 24-hour clock.

Timestamp
A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) designating a date and time.

The *null* value is a special value that is distinct from all non-null values. It means the absence of any other value for that column in the row. The null value exists for all data types.

1. In this book we refer to ISO representations of datetime values.

The following table highlights characteristics of data types used in the examples. All numeric data types are defined in a certain range. The range of numeric data types is also included in this table. You can use this table as a quick reference for proper data type usage.

Data Type	**Type**	**Characteristic**	**Example or Range**
CHAR(15)	fixed- length character string	Maximum length of 254	'Sunny day '
VARCHAR(15)	varying- length character string	Maximum length of 4000	'Sunny day'
SMALLINT	number	2 bytes in length precision of 5 digits	range is -32768 to 32767
INTEGER	number	4 bytes in length precision of 10 digits	range is -2147483648 to 2147483647
REAL	number	single-precision floating point 32 bit approximation	range is -3.402E+38 to -1.175E-37 or 1.175E-37 to -3.402E+38 or zero
DOUBLE	number	double-precision floating point 64 bit approximation	range is -1.79769E+308 to -2.225E-307 or 2.225E-307 to 1.79769E+308 or zero
DECIMAL(5,2)	number	precision is 5 scale is 2	range is -10**31+1 to 10**31-1
DATE	datetime	three-part value	1991-10-27
TIME	datetime	three-part value	13.30.05
TIMESTAMP	datetime	seven-part value	1991-10-27-13.30.05.000000

See the data type compatibility table in the *SQL Reference* for more information.

Chapter 3. Creating Tables and Views

This chapter describes how you can create and manipulate tables and views in DB2 Universal Database. The relationship of tables and views is explored through diagrams and examples.

This chapter covers:

- Creating Tables and Creating Views
- Inserting Data
- Changing Data
- Deleting Data
- Using Views to Manipulate Data

Creating Tables

Create your own tables using the CREATE TABLE statement, specifying the column names and types, as well as *constraints*. Constraints are discussed in "Enforcing Business Rules with Constraints and Triggers" on page 51.

The following statement creates a table named PERS, which is similar to the STAFF table but has an additional column for date of birth.

```
CREATE TABLE PERS
    ( ID             SMALLINT        NOT NULL,
      NAME           VARCHAR(9),
      DEPT           SMALLINT WITH DEFAULT 10,
      JOB            CHAR(5),
      YEARS          SMALLINT,
      SALARY         DECIMAL(7,2),
      COMM           DECIMAL(7,2),
      BIRTH_DATE     DATE)
```

This statement creates a table with no data in it. The next section describes how to insert data into a new table.

As shown in the example, you specify both a name and a data type for each column. Data types are discussed in "Data Types" on page 5. NOT NULL is optional and may be specified to indicate that null values are not allowed in a column. Default values are also optional.

There are many other options you can specify in a CREATE TABLE statement, such as *unique constraints or referential constraints*. For more information about all of the options, see the CREATE TABLE statement in the *SQL Reference*.

Inserting Data

When you create a new table, it does not contain any data. To enter new rows into a table, you use the INSERT statement. This statement has two general forms:

- With one form, you use a VALUES clause to specify values for the columns of one or more rows. The next three examples insert data into tables using this general form.
- With the other form, rather than specifying VALUES, you specify a *fullselect* to identify columns from rows contained in other tables and/or views.

Fullselect is a select statement used in INSERT or CREATE VIEW statements, or following a predicate. A fullselect that is enclosed in parenthesis is commonly referred to as a subquery.

Depending on the default options that you have chosen when creating your table, for every row you insert, you either supply a value for each column or accept a default value. The default values for the various data types are discussed in the *SQL Reference*.

The following statement uses a VALUES clause to insert one row of data into the PERS table:

```
INSERT INTO PERS
   VALUES (12, 'Harris', 20, 'Sales', 5, 18000, 1000, '1950-1-1')
```

The following statement uses a VALUES clause to insert three rows into the PERS table where only the IDs, the names, and the jobs are known. If a column is defined as NOT NULL and it does not have a default value, you must specify a value for it. The NOT NULL clause on a column definition in a CREATE TABLE statement can be extended with the words WITH DEFAULT. If a column is defined as NOT NULL WITH DEFAULT or a constant default such as WITH DEFAULT 10, and you do not specify the column in the column list, the default value is inserted into that column in the inserted row. For example, in the CREATE TABLE statement, a default value was only specified for DEPT column and it was defined to be 10. Hence, the department number (DEPT) is set to 10 and all other columns to null.

```
INSERT INTO PERS (NAME, JOB, ID)
   VALUES ('Swagerman', 'Prgmr', 500),
          ('Limoges', 'Prgmr', 510),
          ('Li', 'Prgmr', 520)
```

The following statement returns the result of the insertions:

```
SELECT *
   FROM PERS

ID     NAME      DEPT   JOB   YEARS  SALARY    COMM      BIRTH_DATE
------ --------- ------ ----- ------ --------- --------- ----------
    12 Harris        20 Sales      5  18000.00   1000.00 01/01/1950
   500 Swagerman     10 Prgmr      -         -         - -
   510 Limoges       10 Prgmr      -         -         - -
   520 Li            10 Prgmr      -         -         - -
```

Note that, in this case, values were not specified for every column. NULL values are displayed as a –. For this to work, the list of column names has to correspond both in order and in data type to the values provided in the VALUES clause. If the list of column names is omitted (as it was in the first example), the list of data values after VALUES must be in the same order as the columns in the table into which they are inserted, and the number of values must equal the number of columns in the table.

Each value must be compatible with the data type of the column into which it is inserted. If a column is defined as nullable and a value for that column is not specified, then the value NULL is given to that column in the inserted row.

The following example inserts the null value into YEARS, COMM and BIRTH_DATE since values have not been specified for those columns in the row.

```
INSERT INTO PERS (ID, NAME, JOB, DEPT, SALARY)
   VALUES (410, 'Perna', 'Sales', 20, 20000)
```

The second form of the INSERT statement is very handy for populating a table with values from rows in another table. As mentioned, rather than specifying VALUES, you specify a fullselect to identify columns from rows contained in other tables and/or views.

The following example selects data from the STAFF table for members of department 38 and inserts it into the PERS table:

```
INSERT INTO PERS (ID, NAME, DEPT, JOB, YEARS, SALARY)
   SELECT ID, NAME, DEPT, JOB, YEARS, SALARY
      FROM STAFF
      WHERE DEPT = 38
```

After this insertion, the following SELECT statement produces a result equal to the fullselect in the INSERT statement.

```
SELECT ID, NAME, DEPT, JOB, YEARS, SALARY
   FROM PERS
   WHERE DEPT = 38
```

The result is:

```
ID     NAME      DEPT   JOB   YEARS  SALARY
------ --------- ------ ----- ------ ---------
    30 Marenghi      38 Mgr       5  17506.75
    40 O'Brien       38 Sales     6  18006.00
    60 Quigley       38 Sales     -  16808.30
   120 Naughton      38 Clerk     -  12954.75
   180 Abrahams      38 Clerk     3  12009.75
```

Changing Data

Use the UPDATE statement to change the data in a table. With this statement, you can change the value of one or more columns in each row that satisfies the search condition of the WHERE clause.

The following example updates information on the employee whose ID is 410:

```
     UPDATE PERS
        SET JOB='Prgmr', SALARY = SALARY + 300
        WHERE ID = 410
```

The SET clause specifies the columns to be updated and provides the values.

The WHERE clause is optional and it specifies the rows to be updated. If the WHERE clause is omitted, the database manager updates each row in the table or view with the values you supply.

In this example, first the table (PERS) is named, then a condition is specified for row that is to be updated. The information for employee number 410, has changed: the employee's job title changed to Prgmr, and the employee's salary increased by $300.

You can change data in more than one row by including a WHERE clause that applies to two or more rows. The following example increases the salary of every salesperson by 15%:

```
     UPDATE PERS
        SET SALARY = SALARY * 1.15
        WHERE JOB = 'Sales'
```

Deleting Data

Use the DELETE statement to delete rows of data from a table based on the search condition specified in the WHERE clause. The following example deletes the row in which the employee ID is 120:

```
DELETE FROM PERS
   WHERE ID = 120
```

The WHERE clause is optional and it specifies the rows to be deleted. If the WHERE clause is omitted, the database manager deletes all rows in the table or view.

You can use the DELETE statement to delete more than one row. The following example deletes all rows in which the employee DEPT is 20:

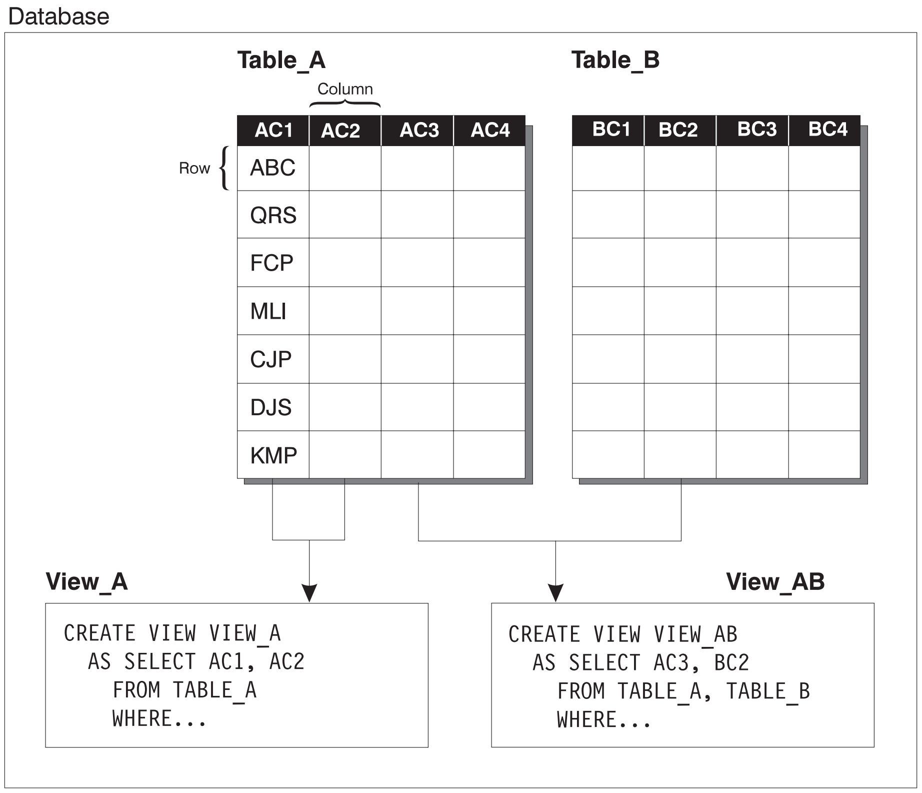
```
DELETE FROM PERS
   WHERE DEPT = 20
```

When you delete a row, you remove the entire row, not specific column values from it.

To delete the definition of a table as well as its contents, issue the DROP TABLE statement as described in the *SQL Reference*.

Creating Views

As discussed in "Views" on page 4, a view provides an alternate way of looking at data in one or more tables. Through creating views, you can restrict the information you want various users to look at. The following diagram shows the relationship between views and tables.



```
CREATE VIEW VIEW_A
  AS SELECT AC1, AC2
    FROM TABLE_A
    WHERE...
```

```
CREATE VIEW VIEW_AB
  AS SELECT AC3, BC2
    FROM TABLE_A, TABLE_B
    WHERE...
```

Figure 2. Relationship Between Tables and Views

In Figure 2, View_A restricts access to only columns AC1 and AC2 of TABLE_A. View_AB allows access to column AC3 in TABLE_A and BC2 in TABLE_B. By creating View_A, you restrict the access users can have to TABLE_A, and by creating VIEW_AB you restrict access to certain columns as well as create an alternate way of looking at the data.

The following statement creates a view of the non-managers in department 20 in the STAFF table, where salary and commission do not show through from the base table.

```
CREATE VIEW STAFF_ONLY
   AS SELECT ID, NAME, DEPT, JOB, YEARS
         FROM STAFF
         WHERE JOB <> 'Mgr' AND DEPT=20
```

After creating the view, the following statement displays the contents of the view:

```
SELECT *
   FROM STAFF_ONLY
```

This statement produces the following result:

```
ID     NAME      DEPT   JOB   YEARS
------ --------- ------ ----- ------
    20 Pernal        20 Sales      8
    80 James         20 Clerk      -
   190 Sneider       20 Clerk      8
```

Earlier, we joined the STAFF and ORG tables to produce a result that listed the name of each department and the name of the manager of that department. The following statement creates a view that can be repetitively used for the same purpose:

```
     CREATE VIEW DEPARTMENT_MGRS
        AS SELECT NAME, DEPTNAME
              FROM STAFF, ORG
              WHERE MANAGER = ID
```

You can put additional constraints on inserts and updates of a table through a view by using the WITH CHECK OPTION clause when you create a view. This clause causes the database manager to validate that any updates of or insertions into the view conform to the view definition, and to reject those that do not. If you omit this clause, inserts and updates are not checked against the view definition. For details on how WITH CHECK OPTION works refer to the CREATE VIEW statement in the *SQL Reference*.

Using Views to Manipulate Data

Like the SELECT statement, INSERT, DELETE, and UPDATE statements can be applied to a view just as though it were a real table. The statements manipulate the data in the underlying base table(s). So when you access the view again, it is evaluated using the most current base table(s). If you do not use the WITH CHECK OPTION, data that you modify using a view may not appear in the repeated accesses of the view, as the data may no longer fit the original view definition.

The following is an example of an update applied to the view FIXED_INCOME:

View Definition for FIXED_INCOME:

```
     CREATE VIEW FIXED_INCOME (LNAME, DEPART, JOBTITLE, NEWSALARY)
        AS SELECT NAME, DEPT, JOB, SALARY
              FROM PERS
              WHERE JOB <> 'Sales' WITH CHECK OPTION

     UPDATE FIXED_INCOME
        SET NEWSALARY = 19000
        WHERE LNAME = 'Li'
```

The update in the previous view is equivalent to (except for the check option) to updating the base table PERS:

```
UPDATE PERS
   SET SALARY = SALARY * 1.10
   WHERE NAME = 'Li'
     AND JOB <> 'Sales'
```

Note that because the view is created using the WITH CHECK OPTION for the constraint `JOB <> 'Sales'` in CREATE VIEW FIXED_INCOME, the following update will not be allowed when Limoges moves over to sales:

```
UPDATE FIXED_INCOME
   SET JOBTITLE = 'Sales'
   WHERE LNAME = 'Limoges'
```

Columns defined by expressions such as SALARY + COMM or SALARY * 1.25 cannot be updated. If a view is defined containing one or more such columns, the owner does not receive the UPDATE privilege on these columns. INSERT statements are not permitted on views containing such columns, but DELETE statements are.

Consider a PERS table with none of the columns defined as NOT NULL. You could insert rows into the PERS table through the FIXED_INCOME view even though it does not contain the ID, YEARS, COMM or BIRTHDATE from underlying table PERS. Columns not visible through the view are set to NULL or the default value, as appropriate.

However, the PERS table does have column ID defined as NOT NULL. If you try to insert a row through the FIXED_INCOME view, the system attempts to insert NULL values into all the PERS columns that are “invisible” through the view. Because the ID column is not included in the view and does not permit null values, the system does not permit the insertion through the view.

For rules and restrictions on modifying views refer to the CREATE VIEW statement in the *SQL Reference*.

Chapter 4. Using SQL Statements to Access Data

This section describes how to connect to a database, and retrieve data using SQL statements.

In the examples, we present the statement to be entered followed in most cases by the results that will be displayed when that statement is issued against the sample database. Note that although we show the statements in uppercase, you can enter them in any mixture of upper and lowercase characters (except where they are enclosed in either single quotes (’) or quotes (’’)).

The SAMPLE database, included with DB2 Universal Database, consists of several tables, as listed in Appendix A. Create the database.

Depending on how your database has been set up, it may be necessary to qualify the table names used, by prefixing them with the schema name and a period. For examples in this book, the default schema is assumed to be USERID. So you could refer to the table ORG as USERID.ORG. Ask your administrator whether or not this is necessary.

This chapter covers:
- Connecting to a Database
- Investigating Errors
- Selecting Columns and Selecting Rows
- Sorting Rows and Removing Duplicate Rows
- Order of Operations
- Using Expressions to Calculate Values
- Naming Expressions
- Selecting Data from More Than One Table
- Using a Subquery
- Using Functions
- Grouping

Connecting to a Database

You need to connect to a database before you can use SQL statements to query or manipulate it. The CONNECT statement associates a database connection with a user name.

For example, to connect to the SAMPLE database, type the following command in the DB2 command line processor :

```
  CONNECT TO SAMPLE USER USERID USING  PASSWORD
```

(Be sure to choose values for USER and USING that are valid on the server system.)

In this example, USER is USERID and USING is PASSWORD.

The following message tells you that you have made a successful connection:

```
        Database Connection Information

           Database product       = DB2/6000 6.0.0
           SQL authorization ID   = USERID
           Local database alias   = SAMPLE
```

When a connection is set through the CONNECT statement an *explicit connection* is established. In an implicit connection the default server has been set. In this case you can use CONNECT or you can just start issuing statements and a connection will automatically be established.

Once you are connected, you can start manipulating the database. For details on implicit and explicit connections refer to the CONNECT statement in the *SQL Reference*.

Investigating Errors

Whenever you make a mistake typing in any of the examples or if an error occurs during execution of an SQL statement, the database manager returns an error message. The error message consists of a message identifier, a brief explanation, and an SQLSTATE.

SQLSTATEs are error codes common to the DB2 family of products. SQLSTATEs conform to the ISO/ANSI SQL92 standard.

For example, if the username or password had been incorrect in the CONNECT statement, the database manager would have returned a message identifier of SQL1403N and an SQLSTATE of 08004. The message is as follows:

```
     SQL1403N The username and/or password supplied is
              incorrect.   SQLSTATE=08004
```

You can get more information about the error message by typing a question mark (?) then the message identifier or the SQLSTATE:

```
        ? SQL1403N
OR
        ? SQL1403
OR
        ? 08004
```

Note that the second last line in the description of the error SQL1403N states that the SQLCODE is -1403. SQLCODE is a produce specific error code. Message identifiers ending with N (Notification) or C (Critical) represent an error and have negative SQLCODEs. Message identifiers ending with W (Warning) represent a warning and have positive SQLCODEs.

Selecting Columns

Use the SELECT statement to select specific columns from a table. In the statement specify a list of column names separated by commas. This list is referred to as a *select list*.

The following statement selects department names (DEPTNAME) and department numbers (DEPTNUMB) from the ORG table of the SAMPLE database:

```
SELECT DEPTNAME, DEPTNUMB
   FROM ORG
```

The above statement produces the following result:

```
DEPTNAME       DEPTNUMB
-------------- --------
Head Office          10
New England          15
Mid Atlantic         20
South Atlantic       38
Great Lakes          42
Plains               51
Pacific              66
Mountain             84
```

By using an asterisk (*) you can select all the columns from the table. The next example lists all columns and rows from the ORG table:

```
SELECT *
   FROM ORG
```

This statement produces the following result:

```
DEPTNUMB DEPTNAME       MANAGER DIVISION   LOCATION
-------- -------------- ------- ---------- -------------
      10 Head Office        160 Corporate  New York
```

```
15 New England         50 Eastern    Boston
20 Mid Atlantic        10 Eastern    Washington
38 South Atlantic      30 Eastern    Atlanta
42 Great Lakes        100 Midwest    Chicago
51 Plains             140 Midwest    Dallas
66 Pacific            270 Western    San Francisco
84 Mountain           290 Western    Denver
```

Selecting Rows

To select specific rows from a table, after the SELECT statement use the WHERE clause to specify the condition or conditions that a row must meet to be selected. A criterion for selecting rows from a table is a *search condition*.

A search condition consists of one or more *predicates*. A predicate specifies a condition that is true or false (or unknown) about a row. You can specify conditions in the WHERE clause by using the following basic predicates:

Predicate	**Function**
x = y	x is equal to y
x <> y	x is not equal to y
x < y	x is less than y
x > y	x is greater than y
x <= y	x is less than or equal to y
x >= y	x is greater than or equal to y
IS NULL / IS NOT NULL	tests for null values

When you construct search conditions, be careful to perform arithmetic operations only on numeric data types, and to make comparisons only among compatible data types. For example, you can't compare strings to numbers.

If you are selecting rows based on a character value, that value must be enclosed in single quotation marks (for example, `WHERE JOB = 'Clerk'`) and each character value must be typed exactly as it exists in the database. If the data value is lowercase in the database and you type it as uppercase, no rows will be selected. If you are selecting rows based on a numeric value, that value must not be enclosed in quotation marks (for example, `WHERE DEPT = 20`).

The following example selects only the rows for department 20 from the STAFF table:

```
SELECT DEPT, NAME, JOB
   FROM STAFF
   WHERE DEPT = 20
```

This statement produces the following result:

```
 DEPT   NAME      JOB
 ------ --------- -----
     20 Sanders   Mgr
     20 Pernal    Sales
     20 James     Clerk
     20 Sneider   Clerk
```

The next example uses AND to specify more than one condition. You can specify as many conditions as you want. The example selects clerks in department 20 from the STAFF table:

```
SELECT DEPT, NAME, JOB
   FROM STAFF
   WHERE JOB = 'Clerk'
   AND DEPT = 20
```

This statement produces the following result:

```
DEPT   NAME      JOB
------ --------- -----
    20 James     Clerk
    20 Sneider   Clerk
```

A null value occurs where no value is entered and the column does not support a default value. It can also occur where the value is specifically set to null. It can occur only in columns that are defined to support null values. Defining and supporting null values in tables are discussed in “Creating Tables” on page 9.

Use the predicate IS NULL, and IS NOT NULL to check for a null value.

The following statement lists employees whose commission is not known:

```
SELECT ID, NAME
   FROM STAFF
   WHERE COMM IS NULL
```

This statement produces the following result:

```
ID     NAME
------ ---------
    10 Sanders
    30 Marenghi
    50 Hanes
   100 Plotz
   140 Fraye
   160 Molinare
   210 Lu
   240 Daniels
   260 Jones
   270 Lea
   290 Quill
```

The value zero is not the same as the null value. The following statement selects everyone in a table whose commission is zero:

```
SELECT ID, NAME
   FROM STAFF
   WHERE COMM = 0
```

Because there are no values of zero in the COMM column in the sample table, the result set returned is empty.

The next example selects all rows where the value of YEARS in the STAFF table is greater than 9:

```
SELECT NAME, SALARY, YEARS
   FROM STAFF
   WHERE YEARS > 9
```

This statement produces the following result:

```
NAME      SALARY    YEARS
--------- --------- ------
Hanes      20659.80     10
Lu         20010.00     10
Jones      21234.00     12
Quill      19818.00     10
Graham     21000.00     13
```

Sorting Rows

You may want the information returned in a specific order. Use the ORDER BY clause to sort the information by the values in one or more columns.

The following statement displays the employees in department 84 ordered by number of years employed:

```
SELECT NAME, JOB, YEARS
   FROM STAFF
   WHERE DEPT = 84
   ORDER BY YEARS
```

This statement produces the following result:

```
NAME      JOB   YEARS
--------- ----- ------
Davis     Sales      5
Gafney    Clerk      5
Edwards   Sales      7
Quill     Mgr       10
```

Specify ORDER BY as the last clause in the entire SELECT statement. Columns named in this clause can be expressions or any column of the table. The column names in the ORDER BY clause do not have to be specified in the select list.

You can order rows in ascending or descending order by explicitly specifying either ASC or DESC within the ORDER BY clause. If neither is specified, the rows are automatically ordered in ascending sequence. The following statement displays the employees in department 84 in descending order by number of years employed:

```
SELECT NAME, JOB, YEARS
   FROM STAFF
   WHERE DEPT = 84
   ORDER BY YEARS DESC
```

This statement produces the following result:

```
NAME      JOB   YEARS
--------- ----- ------
Quill     Mgr       10
Edwards   Sales      7
Davis     Sales      5
Gafney    Clerk      5
```

You can order rows by character values as well as numeric values. The following statement displays the employees in department 84 in alphabetical order by name:

```
SELECT NAME, JOB, YEARS
   FROM STAFF
   WHERE DEPT = 84
   ORDER BY NAME
```

This statement produces the following result:

```
NAME      JOB   YEARS
--------- ----- ------
Davis     Sales      5
Edwards   Sales      7
Gafney    Clerk      5
Quill     Mgr       10
```

Removing Duplicate Rows

When using the SELECT statement, you may not want duplicate information to be returned. For example, STAFF has a DEPT column in which several department numbers are listed more than once, and a JOB column in which several job descriptions are listed more than once.

To eliminate duplicate rows, use the DISTINCT option on the SELECT clause. For example, if you insert DISTINCT into the statement, each job within a department is listed only once:

```
SELECT DISTINCT DEPT, JOB
   FROM STAFF
   WHERE DEPT < 30
   ORDER BY DEPT, JOB
```

This statement produces the following result:

```
DEPT   JOB
------ -----
    10 Mgr
    15 Clerk
    15 Mgr
    15 Sales
    20 Clerk
    20 Mgr
    20 Sales
```

DISTINCT has eliminated all rows that contain duplicate data in the set of columns specified in the SELECT statement.

Order of Operations

It is important to take into accout the order of operations. Output of one clause is the input to the next one as stated in the list below. An example where order of operations is a consideration is presented in “Naming Expressions” on page 25.

Also, note that this explanation allows for a more intuitive way of thinking about queries. It is not necessarily the way the operations are performed internally. The sequence of operations is as follows:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause

Using Expressions to Calculate Values

An *expression* is a calculation or function that you include in a statement. The following statement calculates what the salaries for each employee in department 38 would be if each received a $500 bonus:

```
SELECT DEPT, NAME, SALARY + 500
   FROM STAFF
   WHERE DEPT = 38
   ORDER BY 3
```

This result is:

```
DEPT   NAME      3
------ --------- ----------------
    38 Abrahams          12509.75
    38 Naughton          13454.75
    38 Quigley           17308.30
    38 Marenghi          18006.75
    38 O'Brien           18506.00
```

Note that the column name for the third column is a number. This is a system generated number, since SALARY+500 does not specify a column name. Later on this number is used in the ORDER BY clause to refer to the third column. "Naming Expressions" talks about how to give meaningful names to expressions.

You can form arithmetic expressions using the basic arithmetic operators for addition (+), subtraction (−), multiplication (*) and division (/).

The operators can operate on values from several different types of operands, some of which are:

- Column names (as in RATE * HOURS)
- Constant values (as in RATE * 1.07)
- Scalar functions (as in LENGTH(NAME) + 1).

Naming Expressions

The optional AS clause lets you assign a meaningful name to an expression, which makes referring back to the expression easier. You can use an AS clause to provide a name for any item in the select list.

The following statement displays all employees whose salary plus commission is less than $13, 000. The expression SALARY + COMM is named PAY:

```
SELECT NAME, JOB, SALARY + COMM AS PAY
   FROM STAFF
   WHERE (SALARY + COMM) < 13000
   ORDER BY PAY
```

This statement produces the following result:

```
NAME      JOB   PAY
--------- ----- ----------
Yamaguchi Clerk   10581.50
Burke     Clerk   11043.50
Scoutten  Clerk   11592.80
Abrahams  Clerk   12246.25
Kermisch  Clerk   12368.60
Ngan      Clerk   12714.80
```

By using the AS clause, you can refer to a particular column name rather than the system generated number in the ORDER BY clause. In this example we compare (SALARY + COMM) with 13000 in the WHERE clause, instead of using the name PAY. This is a result of the order of operations. The WHERE clause is evaluated before (SALARY + COMM) is given the name PAY. Hence, PAY cannot be used in the predicate.

Selecting Data from More Than One Table

You can use the SELECT statement to produce reports that contain information from two or more tables. This is commonly referred to as a *join*. For example, you can join data from the STAFF and ORG tables to form a new table. To join two tables, specify the columns you want to be displayed in the SELECT clause, the table names in a FROM clause and the search condition in the WHERE clause. The WHERE clause is optional.

The next example associates the name of each manager with a department name. You need to select information from two tables since the employee information (STAFF table) and the departmental information (ORG table) are stored separately. The following query selects the NAME and DEPTNAME columns for STAFF and ORG tables, respectively. The search condition narrows down the selection to rows where the values in the MANAGER column are the same as the values in the ID column:

```
SELECT DEPTNAME, NAME
   FROM ORG, STAFF
   WHERE MANAGER = ID
```

Figure 3 on page 27 demonstrates how columns in two different tables are compared. The boxed values indicate a match where the search condition has been satisfied.

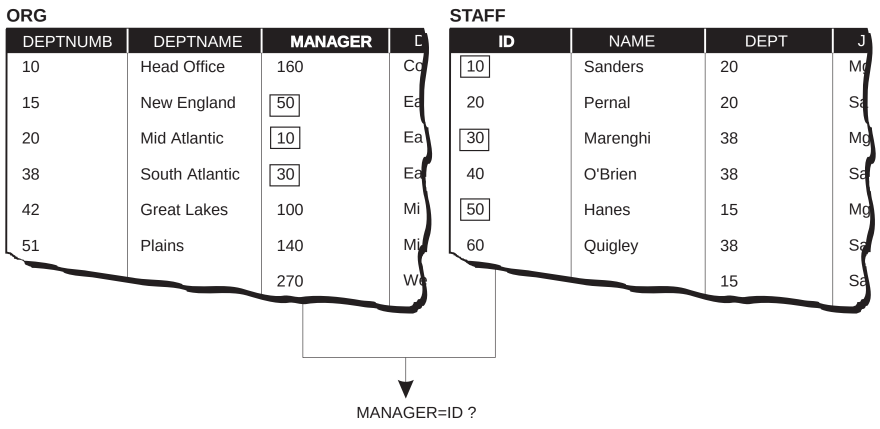


Figure 3. Selecting from STAFF and ORG tables

The SELECT statement produces the following result:

```
DEPTNAME       NAME
-------------- ---------
Mid Atlantic   Sanders
South Atlantic Marenghi
New England    Hanes
Great Lakes    Plotz
Plains         Fraye
Head Office    Molinare
Pacific        Lea
Mountain       Quill
```

The result lists the name of each manager and his or her department.

Using a Subquery

When you write a SELECT statement, you can place another SELECT statement within the WHERE clause. Each additional SELECT starts a *subquery*.

A subquery can, in turn, include another subquery whose value is substituted into its WHERE clause. In addition, a WHERE clause can include subqueries in more than one search condition. The subquery can refer to tables and columns that are different than the ones used in the main query.

The following statement selects the division and location from the ORG table of the employee whose ID in the STAFF table is 280:

```
SELECT DIVISION, LOCATION
   FROM ORG
   WHERE DEPTNUMB = (SELECT DEPT
                        FROM STAFF
                        WHERE ID = 280)
```

When processing this statement, DB2 first determines the result of the subquery. The result is 66, since the employee with ID 280 is in department 66. Then the final result is taken from the row of the ORG table whose DEPTNUMB column has the value of 66. The final result is:

```
DIVISION   LOCATION
---------- -------------
Western    San Francisco
```

When you use a subquery, the database manager evaluates it and substitutes the resulting value directly into the WHERE clause.

Subqueries are further discussed "Correlated Subqueries" on page 39.

Using Functions

This section gives you a brief introduction to functions that will be used in the examples throughout the book. A *database function* is a relationship between a set of input data values and a result value.

Functions can be either *built-in* or *user-defined*. DB2 Universal Database delivers many built-in and preinstalled user-defined functions. You can find the built-in functions in the SYSIBM schema and the preinstalled user-defined functions in the SYSFUN schema. SYSIBM and SYSFUN are reserved schemas.

The built-in and preinstalled user-defined functions will never satisfy all requirements. So application developers may need to create their own suite of functions specific to their applications. User-defined functions make this possible, expanding the scope of DB2 Universal Database to include, for example, customized business or scientific functions. This is further discussed in the "User-Defined Functions" on page 66.

Column Functions

Column functions operate on a set of values in a column to derive a single result value. The following are just a few examples of column functions. For a full list refer to the *SQL Reference*.

AVG Returns the sum of the values in a set divided by the number of values in that set

COUNT	Returns the number of rows or values in a set of rows or values
MAX	Returns the largest value in a set of values
MIN	Returns the smallest value in a set of values

The following statement selects the maximum salary from the STAFF table:

```
SELECT MAX(SALARY)
   FROM STAFF
```

This statement returns the value 22959.20 from the STAFF sample table.

The next example selects the names and salaries of employees whose income is more than the average income yet have been with the company less than the average number of years.

```
SELECT NAME, SALARY
   FROM STAFF
   WHERE SALARY > (SELECT AVG(SALARY) FROM STAFF)
   AND YEARS < (SELECT AVG(YEARS) FROM STAFF)
```

This statement produces the following result:

```
 NAME      SALARY
--------- ---------
Marenghi   17506.75
Daniels    19260.25
Gonzales   16858.20
```

In the above example, in the WHERE clause, the column function is stated in a subquery as opposed to being directly implemented (WHERE SALARY > AVG(SALARY)). Column functions cannot be stated in the WHERE clause. This is due to the order of operations. The WHERE clause can be thought of as being evaluated before the SELECT clause. Consequently, when the WHERE clause is being evaluated, the column function does not have access to the set of values. This set of values are selected at a later time by the SELECT clause.

You can specify DISTINCT as part of the argument of a column function to eliminate duplicate values before a function is applied. Thus, COUNT(DISTINCT WORKDEPT) computes the number of different departments.

Scalar Functions

A *scalar function* performs some operation on a value to return another value. The following are just a few examples of scalar functions provided by DB2 Universal Database.

ABS	Return the absolute value of a number

HEX	Returns the hexadecimal representation of a value
LENGTH	Returns the number of bytes in an argument (for a graphic string it returns the number of double-byte characters.)
YEAR	Extract the year portion of a datetime value

For a detailed list and description of scalar functions refer to the *SQL Reference*.

The following statement returns the department names from the ORG table together with the length of each of these names:

```
SELECT DEPTNAME, LENGTH(DEPTNAME)
   FROM ORG
```

This statement produces the following result:

```
DEPTNAME       2
-------------- -----------
Head Office             11
New England             11
Mid Atlantic            12
South Atlantic          14
Great Lakes             11
Plains                   6
Pacific                  7
Mountain                 8
```

Note that since the AS clause was not used to give a meaningful name to LENGTH(DEPTNAME), a system generated number appears in the second column.

Grouping

DB2 Universal Database has the capability of analyzing data based on particular columns of a table.

You can group rows according to the group defined in a GROUP BY clause. In its simplest form, a group consists of columns known as *grouping columns*. The column names in the SELECT clause must be either a grouping column or a column function. Column functions return a result for each group defined by the GROUP BY clause. The following example produces a result that lists the maximum salary for each department number:

```
SELECT DEPT, MAX(SALARY) AS MAXIMUM
   FROM STAFF
   GROUP BY DEPT
```

This statement produces the following result:

```
DEPT   MAXIMUM
------ ---------
    10  22959.20
    15  20659.80
    20  18357.50
    38  18006.00
    42  18352.80
    51  21150.00
    66  21000.00
    84  19818.00
```

Note that the MAX(SALARY) is calculated for each department, a group defined by the GROUP BY clause, not the entire company.

Using a WHERE Clause with a GROUP BY Clause

A grouping query can have a standard WHERE clause that eliminates non-qualifying rows before the groups are formed and the column functions are computed. You have to specify the WHERE clause *before* the GROUP BY clause. For example:

```
SELECT WORKDEPT, EDLEVEL, MAX(SALARY) AS MAXIMUM
   FROM EMPLOYEE
   WHERE HIREDATE > '1979-01-01'
   GROUP BY WORKDEPT, EDLEVEL
   ORDER BY WORKDEPT, EDLEVEL
```

The result is:

```
WORKDEPT EDLEVEL MAXIMUM
-------- ------- -----------
D11           17    18270.00
D21           15    27380.00
D21           16    36170.00
D21           17    28760.00
E11           12    15340.00
E21           14    26150.00
```

Note that every column name specified in the SELECT statement is also mentioned in the GROUP BY clause. Not mentioning the column names in both places will give you an error. The GROUP BY clause returns a row for each unique combination of WORKDEPT and EDLEVEL.

Using the HAVING Clause After the GROUP BY Clause

You can apply a qualifying condition to groups so that the system returns a result only for the groups that satisfy the condition. To do this, include a HAVING clause *after* the GROUP BY clause. A HAVING clause can contain one or more predicates connected by ANDs and ORs. Each predicate compares a property of the group (such as AVG(SALARY)) with either:

- Another property of the group

For example:

```
HAVING AVG(SALARY) > 2 * MIN(SALARY)
```

- A constant

  For example:

  ```
   HAVING AVG(SALARY) > 20000
  ```

For example, the following query finds the maximum and minumum salary of departments with more than 4 employees:

```
SELECT WORKDEPT, MAX(SALARY) AS MAXIMUM, MIN(SALARY) AS MINIMUM
   FROM EMPLOYEE
   GROUP BY WORKDEPT
   HAVING COUNT(*) > 4
   ORDER BY WORKDEPT
```

This statement produces the following result:

```
WORKDEPT MAXIMUM     MINIMUM
-------- ----------- -----------
D11         32250.00    18270.00
D21         36170.00    17250.00
E11         29750.00    15340.00
```

It is possible (though unusual) for a query to have a HAVING clause but no GROUP BY clause. In this case, DB2 treats the entire table as one group. Because the table is treated as a single group, you can have at most one result row. If the HAVING condition is true for the table as a whole, the selected result (which must consist entirely of column functions) is returned; otherwise no rows are returned.

Chapter 5. Expressions and Subqueries

DB2 provides flexibility in expressing queries. This chapter describes a few of the important methods available in expressing more complex queries.

This chapter gives a comprehensive description of the following:
- Scalar Fullselects
- Casting Data Types
- Case Expressions
- Table Expressions
- Correlation Names

Scalar Fullselects

A scalar fullselect is a fullselect within parentheses that returns one row containing only one column value. Scalar fullselects are useful for retrieving data values from the database for use in an expression.
- The following example lists names of employees who have a salary greater than the average salary of all employees:

```
SELECT LASTNAME, FIRSTNME
   FROM EMPLOYEE
   WHERE  SALARY > (SELECT AVG(SALARY)
                      FROM EMPLOYEE)
```

- This example finds the average salary of the employees in two different tables:

```
SELECT AVG(SALARY) AS "Average_Employee",
       (SELECT AVG(SALARY) AS "Average_Staff" FROM STAFF)
           FROM EMPLOYEE
```

Casting Data Types

There may be times when you need to convert values from one data type to another, for example, from a numeric value to a character string. To convert a value to a different type, use the CAST specification.

Another possible use for a cast specification is to truncate a very long character string. In the EMP_RESUME table the column RESUME is CLOB(5K). You may want to display only the first 370 characters containing the personal information of the applicant. To display the first 370 characters of the ASCII format of the resumes from the table EMP_RESUME, issue the following query:

```
SELECT EMPNO, CAST(RESUME AS VARCHAR(370))
   FROM EMP_RESUME
   WHERE RESUME_FORMAT = 'ascii'
```

A warning is issued informing you that values longer than 370 characters are truncated.

You can cast NULL values to other data types that are more convenient for manipulation in a query. “Common Table Expressions” on page 36 is an example of using casting for this purpose.

Case Expressions

You can use CASE expressions in SQL statements to easily manipulate the data representation of a table. This provides a powerful conditional expression capability that is similar in concept to CASE statements in some programming languages.

- To change department numbers from the DEPTNAME column in ORG table to meaningful words, enter the following query:

  ```
  SELECT DEPTNAME,
     CASE DEPTNUMB
       WHEN 10 THEN 'Marketing'
       WHEN 15 THEN 'Research'
       WHEN 20 THEN 'Development'
       WHEN 38 THEN 'Accounting'
       ELSE 'Sales'
     END  AS FUNCTION
     FROM ORG
  ```

  The result is:

  ```
  DEPTNAME       FUNCTION
  -------------- -----------
  Head Office    Marketing
  New England    Research
  Mid Atlantic   Development
  South Atlantic Accounting
  Great Lakes    Sales
  Plains         Sales
  Pacific        Sales
  Mountain       Sales
  ```

- You can use CASE expressions to protect against exceptions such as division by zero. In the following example, if the employee has no bonus or commission payment, the statement condition prevents an error by avoiding the division operation:

```
SELECT LASTNAME, WORKDEPT FROM EMPLOYEE
   WHERE(CASE
            WHEN BONUS+COMM=0 THEN NULL
            ELSE SALARY/(BONUS+COMM)
         END ) > 10
```

- You can use a CASE expression to produce a ratio based on the sum of a subset of values from one column to the sum of all the values from that column in a single statement. A statement using a CASE expression requires only a single pass through the data. Without a CASE expression, at least two passes are required to perform the same calculation.

  The following example computes the ratio of the sum of the salaries of department 20 to the total of all salaries using a CASE expression:

```
SELECT CAST(CAST (SUM(CASE
                         WHEN DEPT = 20 THEN SALARY
                         ELSE 0
                      END) AS DECIMAL(7,2))/
                  SUM(SALARY) AS DECIMAL (3,2))
   FROM STAFF
```

  The result is 0.11. Note that the CAST functions ensure that the precision of the result is preserved.

- You can use a CASE expression to evaluate a simple function instead of calling the function itself, which would require additional overhead. For example:

```
CASE
   WHEN X<0 THEN -1
   WHEN X=0 THEN 0
   WHEN X>0 THEN 1
END
```

  This expression has the same result as the SIGN user-defined function in the SYSFUN schema.

Table Expressions

If you just need the definition of a view for a single query, you can use a *table expression*.

Table expressions are temporary and are only valid for the life of the SQL statement; they cannot be shared, but they allow more flexibility than views. View definitions can be shared by any authorized user.

This section describes how to use common table expressions and nested table expressions in queries.

Nested Table Expressions

A nested table expression is a temporary view where the definition is *nested* (defined directly) in the FROM clause of the main query.

The following query uses a nested table expression to find the average total pay, education level and year of hire, for those with an education level greater than 16:

```
SELECT EDLEVEL, HIREYEAR, DECIMAL(AVG(TOTAL_PAY), 7,2)
   FROM (SELECT YEAR(HIREDATE) AS HIREYEAR, EDLEVEL,
                SALARY+BONUS+COMM AS TOTAL_PAY
            FROM EMPLOYEE
            WHERE EDLEVEL > 16 )  AS PAY_LEVEL
   GROUP BY EDLEVEL, HIREYEAR
   ORDER BY EDLEVEL, HIREYEAR
```

The result is as follows:

```
EDLEVEL HIREYEAR    3
------- ----------- ---------
     17        1967  28850.00
     17        1973  23547.00
     17        1977  24430.00
     17        1979  25896.50
     18        1965  57970.00
     18        1968  32827.00
     18        1973  45350.00
     18        1976  31294.00
     19        1958  51120.00
     20        1975  42110.00
```

This query uses a nested table expression to first extract the year of hire from the HIREDATE column so that it can subsequently be used in the GROUP BY clause. You may not want to create this as a view, because you intend to perform similar queries using different values for EDLEVEL.

The scalar built-in function DECIMAL is used in this example. DECIMAL returns a decimal representation of a number or a character string. For more details on functions refer to the *SQL Reference*.

Common Table Expressions

A *common table expression* is a named result table that is defined using the WITH keyword prior to the beginning of a fullselect. It is a table expression that you create to use throughout a complex query. Define and name it at the start of the query using a WITH clause. Repeated references to a common table expression use the same result set. By comparison, if you used nested table expressions or views, the result set would be regenerated each time, with possibly different results.

The following example lists all the people in the company who have an education level greater than 16, who make less pay on average than those people who were hired at the same time and who have the same education. The parts of the query are described in further detail following the query.

1

```
WITH
      PAYLEVEL AS
         (SELECT EMPNO, YEAR(HIREDATE) AS HIREYEAR, EDLEVEL,
                 SALARY+BONUS+COMM AS TOTAL_PAY
             FROM EMPLOYEE
             WHERE EDLEVEL > 16),
```

2

```
      PAYBYED (EDUC_LEVEL, YEAR_OF_HIRE, AVG_TOTAL_PAY) AS
         (SELECT EDLEVEL, HIREYEAR, AVG(TOTAL_PAY)
             FROM PAYLEVEL
             GROUP BY EDLEVEL, HIREYEAR)
```

3

```
SELECT EMPNO, EDLEVEL, YEAR_OF_HIRE, TOTAL_PAY, DECIMAL(AVG_TOTAL_PAY,7,2)
   FROM PAYLEVEL, PAYBYED
   WHERE EDLEVEL=EDUC_LEVEL
     AND HIREYEAR = YEAR_OF_HIRE
     AND TOTAL_PAY < AVG_TOTAL_PAY
```

1 This is a common table expression with the name PAYLEVEL. This result table includes the year that a person was hired, the total pay for that employee, and his or her education level. Only rows for employees with an education level greater than 16 are included.

2 This is a common table expression with the name PAYBYED (or PAY by education). It uses the PAYLEVEL table that was created in the previous common table expression to determine the education level, hire year, and average pay of employees within each education level, hired in the same year. The columns returned by this table have been given different names (EDUC_LEVEL, for example) from the column names used in the select list. This produces a result set named PAYBYED that is the same as the result produced in the nested table expression example.

3 Finally, we get to the actual query that produces the desired result. The two tables (PAYLEVEL, PAYBYED) are joined to determine those individuals who have total pay that is less than the average pay for people hired in the same year. Note that PAYBYED is based on

PAYLEVEL. So PAYLEVEL is effectively accessed twice in the complete statement. Both times the same set of rows are used in evaluating the query.

The final result is as follows:

```
EMPNO  EDLEVEL YEAR_OF_HIRE TOTAL_PAY     5
------ ------- ------------ ------------ ---------
000210      17         1979      20132.00  25896.50
```

Correlation Names

A *correlation name* is an identifier used for distinguishing multiple uses of an object. A correlation name can be defined in the FROM clause of a query and in the first clause of an UPDATE or DELETE statement. It can be associated with a table, view, or a nested table expression but only within the context that it is defined.

For example, the clause FROM STAFF S, ORG O establishes S and O as the correlation names for STAFF and ORG, respectively.

```
 SELECT NAME, DEPTNAME
    FROM STAFF S, ORG O
    WHERE O.MANAGER = S.ID
```

Once you have defined a correlation name, you can only use the correlation name to qualify the object. For example, in the example above had we stated ORG.MANAGER=STAFF.ID the statement would have failed.

You can also use a correlation name as a shorter name for referring to a database object. Typing just S is easier than typing STAFF.

By using correlation names, you can make duplicates of an object. This is useful when you need to compare entries of a table with itself. In the following example, table EMPLOYEE is compared with another instance of itself to find the managers of all employees. It displays the name of the employees who are not designers, name of their manager and the department number.

```
SELECT E2.FIRSTNME, E2.LASTNAME,
       E2.JOB, E1.FIRSTNME, E1.LASTNAME, E1.WORKDEPT
   FROM EMPLOYEE E1, EMPLOYEE E2
   WHERE E1.WORKDEPT = E2.WORKDEPT
     AND E1.JOB = 'MANAGER'
     AND E2.JOB <> 'MANAGER'
     AND E2.JOB <> 'DESIGNER'
```

This statement produces the following result:

```
FIRSTNME     LASTNAME        JOB      FIRSTNME     LASTNAME        WORKDEPT
------------ --------------- -------- ------------ --------------- --------
DOLORES      QUINTANA        ANALYST  SALLY        KWAN            C01
HEATHER      NICHOLLS        ANALYST  SALLY        KWAN            C01
JAMES        JEFFERSON       CLERK    EVA          PULASKI         D21
MARIA        PEREZ           CLERK    EVA          PULASKI         D21
SYBIL        JOHNSON         CLERK    EVA          PULASKI         D21
DANIEL       SMITH           CLERK    EVA          PULASKI         D21
SALVATORE    MARINO          CLERK    EVA          PULASKI         D21
ETHEL        SCHNEIDER       OPERATOR EILEEN       HENDERSON       E11
MAUDE        SETRIGHT        OPERATOR EILEEN       HENDERSON       E11
PHILIP       SMITH           OPERATOR EILEEN       HENDERSON       E11
JOHN         PARKER          OPERATOR EILEEN       HENDERSON       E11
RAMLAL       MEHTA           FIELDREP THEODORE     SPENSER         E21
JASON        GOUNOT          FIELDREP THEODORE     SPENSER         E21
WING         LEE             FIELDREP THEODORE     SPENSER         E21
```

Correlated Subqueries

A subquery that is allowed to refer to any of the previously mentioned tables is known as a *correlated subquery*. We also say that the subquery has a *correlated reference* to a table in the main query.

The following example is an uncorrelated subquery that lists the employee number and name of employees in department 'A00' with a salary greater than the average salary of the department:

```
SELECT EMPNO, LASTNAME
   FROM EMPLOYEE
   WHERE WORKDEPT = 'A00'
     AND SALARY > (SELECT AVG(SALARY)
                     FROM EMPLOYEE
                     WHERE WORKDEPT = 'A00')
```

If you want to know the average salary for every department, the subquery needs to be evaluated once for every department. You can do this by using the correlation capability of SQL, which permits you to write a subquery that is executed repeatedly, once for each row of the table identified in the outer-level query. This type of correlated subquery is used to compute some property of each row of the outer-level table that is needed to evaluate a predicate in the subquery.

This example shows all the employees whose salary is higher than the average salary of their department:

```
SELECT E1.EMPNO, E1.LASTNAME, E1.WORKDEPT
   FROM EMPLOYEE E1
   WHERE SALARY > (SELECT AVG(SALARY)
                     FROM EMPLOYEE E2
                     WHERE E2.WORKDEPT = E1.WORKDEPT)
   ORDER BY E1.WORKDEPT
```

In this query, the subquery is evaluated once for every department. The result is:

```
EMPNO  LASTNAME        WORKDEPT
------ --------------- --------
000010 HAAS            A00
000110 LUCCHESSI       A00
000030 KWAN            C01
000060 STERN           D11
000220 LUTZ            D11
000200 BROWN           D11
000170 YOSHIMURA       D11
000150 ADAMSON         D11
000070 PULASKI         D21
000270 PEREZ           D21
000240 MARINO          D21
000090 HENDERSON       E11
000280 SCHNEIDER       E11
000100 SPENSER         E21
000340 GOUNOT          E21
000330 LEE             E21
```

To write a query with a correlated subquery, use the same basic format of an ordinary outer query with a subquery. However, in the FROM clause of the outer query, just after the table name, place a correlation name. The subquery may then contain column references qualified by the correlation name. For example, if E1 is a correlation name, then E1.WORKDEPT means the WORKDEPT value of the current row of the table in the outer query. The subquery is (conceptually) reevaluated for each row of the table in the outer query.

By using a correlated subquery, you let the system do the work for you and reduce the amount of code you need to write within your application.

Unqualified correlated references are allowed in DB2. For example, the table EMPLOYEE has a column named LASTNAME and table SALES has a column named SALES_PERSON, but no column named LASTNAME.

```
SELECT LASTNAME, FIRSTNME, COMM
   FROM EMPLOYEE
   WHERE 3 > (SELECT AVG(SALES)
                FROM SALES
                WHERE LASTNAME = SALES_PERSON)
```

In this example, the system checks the innermost FROM clause for a LASTNAME column. Not finding one, it then checks the next innermost

FROM clause (which in this case is the outer FROM clause). While not always necessary, qualifying correlated references is recommended to improve the readability of the query and to ensure that you are getting the result that you intend.

Implementing a Correlated Subquery

When would you want to use a correlated subquery? The use of a column function is sometimes a clue.

Let's say you want to list the employees whose level of education is higher than the average for their department.

First, you must determine the select-list items. The problem says "List the employees". This implies that the EMPNO from the EMPLOYEE table should be sufficient to uniquely identify employees. The problem also states the level of education (EDLEVEL) and the employees' departments (WORKDEPT) as conditions. While the problem does not explicitly ask for columns to be displayed, including them in the select-list will help illustrate the solution. A part of the query can now be constructed:

```
SELECT LASTNAME, WORKDEPT, EDLEVEL
   FROM EMPLOYEE
```

Next, a search condition (WHERE clause) is needed. The problem statement says, "...whose level of education is higher than the average for that employee's department". This means that for every employee in the table, the average education level for that employee's department must be computed. This statement fits the description of a correlated subquery. Some property (average level of education of the current employee's department) is being computed for each row. A correlation name is needed for the EMPLOYEE table:

```
SELECT LASTNAME, WORKDEPT, EDLEVEL
   FROM EMPLOYEE E1
```

The subquery needed is simple. It computes the average level of education for each department. The complete SQL statement is:

```
SELECT LASTNAME, WORKDEPT, EDLEVEL
   FROM EMPLOYEE E1
   WHERE EDLEVEL > (SELECT AVG(EDLEVEL)
                     FROM EMPLOYEE  E2
                     WHERE E2.WORKDEPT = E1.WORKDEPT)
```

The result is:

```
LASTNAME        WORKDEPT EDLEVEL
--------------- -------- -------
HAAS            A00           18
KWAN            C01           20
```

```
     PULASKI        D21            16
     HENDERSON      E11            16
     LUCCHESSI      A00            19
     PIANKA         D11            17
     SCOUTTEN       D11            17
     JONES          D11            17
     LUTZ           D11            18
     MARINO         D21            17
     JOHNSON        D21            16
     SCHNEIDER      E11            17
     MEHTA          E21            16
     GOUNOT         E21            16
```

Suppose that instead of listing the employee's department number, you list the department name. The information you need (DEPTNAME) is in a separate table (DEPARTMENT). The outer-level query that defines a correlation variable can also be a join query (see "Selecting Data from More Than One Table" on page 26 for details).

When you use joins in an outer-level query, list the tables to be joined in the FROM clause, and place the correlation name next to any of these table names.

To modify the query to list the department's name instead of its number, replace WORKDEPT by DEPTNAME in the select-list. The FROM clause must now also include the DEPARTMENT table, and the WHERE clause must express the appropriate join condition.

This is the modified query:

```
     SELECT LASTNAME, DEPTNAME, EDLEVEL
        FROM EMPLOYEE E1, DEPARTMENT
        WHERE E1.WORKDEPT = DEPARTMENT.DEPTNO
        AND EDLEVEL > (SELECT AVG(EDLEVEL)
                          FROM EMPLOYEE E2
                          WHERE E2.WORKDEPT = E1.WORKDEPT)
```

The above examples show that the correlation name used in a subquery must be defined in the FROM clause of some query that contains the correlated subquery. However, this containment may involve several levels of nesting.

Suppose that some departments have only a few employees and therefore their average education level may be misleading. You might decide that in order for the average level of education to be a meaningful number to compare an employee against, there must be at least five employees in a department. So now we have to list the employees whose level of education is higher than the average for that employee's department, and only consider departments with at least five employees.

The problem implies another subquery because, for each employee in the outer-level query, the total number of employees in that person's department must be counted:

```
SELECT COUNT(*)
   FROM EMPLOYEE E3
   WHERE E3.WORKDEPT = E1.WORKDEPT
```

Only if the count is greater than or equal to 5 is an average to be computed:

```
SELECT AVG(EDLEVEL)
   FROM EMPLOYEE E2
   WHERE E2.WORKDEPT = E1.WORKDEPT
   AND 5 <= (SELECT COUNT(*)
                FROM EMPLOYEE  E3
                WHERE E3.WORKDEPT = E1.WORKDEPT)
```

Finally, only those employees whose level of education is greater than the average for that department are included:

```
SELECT LASTNAME, DEPTNAME, EDLEVEL
   FROM EMPLOYEE E1, DEPARTMENT
   WHERE E1.WORKDEPT = DEPARTMENT.DEPTNO
   AND EDLEVEL >
   (SELECT AVG(EDLEVEL)
       FROM EMPLOYEE E2
       WHERE E2.WORKDEPT = E1.WORKDEPT
       AND 5 <=
       (SELECT COUNT(*)
           FROM EMPLOYEE E3
           WHERE E3.WORKDEPT = E1.WORKDEPT))
```

This statement produces the following result:

```
LASTNAME        DEPTNAME                      EDLEVEL
--------------- ----------------------------- -------
PIANKA          MANUFACTURING SYSTEMS              17
LUTZ            MANUFACTURING SYSTEMS              18
JONES           MANUFACTURING SYSTEMS              17
SCOUTTEN        MANUFACTURING SYSTEMS              17
PULASKI         ADMINISTRATION SYSTEMS             16
JOHNSON         ADMINISTRATION SYSTEMS             16
MARINO          ADMINISTRATION SYSTEMS             17
HENDERSON       OPERATIONS                         16
SCHNEIDER       OPERATIONS                         17
```

Chapter 6. Using Operators and Predicates in Queries

In DB2 Universal Database you can combine queries with different *set operators* and construct complex conditional statements with *quantified predicates*.

This chapter explains how to:
- Combine different tables with UNION, EXCEPT and INTERSECT set operators
- Construct complex conditions for queries with quantified predicates. Basic predicates were discussed briefly in “Selecting Rows” on page 20.

Combining Queries by Set Operators

The UNION, EXCEPT, and INTERSECT set operators enable you to combine two or more outer-level queries into a single query. Each of the queries connected by these set operators is executed and the individual results are combined. Depending on the operator, a different result is produced.

UNION Operator

The UNION operator derives a result table by combining two other result tables (for example TABLE1 and TABLE2) and eliminating any duplicate rows in the tables. When ALL is used with UNION (that is, UNION ALL), duplicate rows are not eliminated. In either case, each row of the derived table is a row from either TABLE1 or TABLE2.

In the following example of the UNION operator, the query returns the names of all persons that have a salary greater than $21, 000, or that have managerial responsibilities and have been working for less than 8 years:

1

```
SELECT ID, NAME FROM STAFF WHERE SALARY > 21000
UNION
```

2

```
SELECT ID, NAME FROM STAFF WHERE JOB='Mgr' AND YEARS < 8
ORDER BY ID
```

The result of the individual queries are as follows:

1

```
ID     NAME
------ ---------
   140 Fraye
   160 Molinare
   260 Jones
```

2

```
ID     NAME
------ ---------
    10 Sanders
    30 Marenghi
   100 Plotz
   140 Fraye
   160 Molinare
   240 Daniels
```

The database manager combines the results of both queries, eliminates the duplicates, and returns the final result in ascending order.

```
ID     NAME
------ ---------
    10 Sanders
    30 Marenghi
   100 Plotz
   140 Fraye
   160 Molinare
   240 Daniels
   260 Jones
```

If you use the ORDER BY clause in a query with any set operator, you must write it after the last query. The system applies the ordering to the combined answer set. If the column name in the two tables is different, the combined result table does not have names for the corresponding columns. Instead, the columns are numbered in the order in which they appear. So, if you want the result table to be ordered, you have to specify the column number in the ORDER BY clause.

EXCEPT Operator

The EXCEPT operator derives a result table by including all rows that are in TABLE1 but not in TABLE2, and eliminating all duplicate rows. When you use ALL with EXCEPT (EXCEPT ALL), the duplicate rows are not eliminated.

In the following example of the EXCEPT operator, the query returns the names of all persons that earn over $21, 000 but do not have the position of a manager and have been there 8 years or more.

```
SELECT ID, NAME FROM STAFF WHERE SALARY > 21000
EXCEPT
SELECT ID, NAME FROM STAFF WHERE JOB='Mgr' AND YEARS < 8
```

The result of the individual queries is listed in the section on UNION. The above statement produces the following result:

```
ID     NAME
------ ---------
   260 Jones
```

INTERSECT operator

The INTERSECT operator derives a result table by including only rows that exist in both TABLE1 and TABLE2 and eliminating all duplicate rows. When you use ALL with INTERSECT (INTERSECT ALL), the duplicate rows are not eliminated.

In the following example of the INTERSECT operator, the query returns the name and ID of employees that earn more than $21, 000, have managerial responsibilites and have been working for fewer than 8 years.

```
SELECT ID, NAME FROM STAFF WHERE SALARY > 21000
INTERSECT
SELECT ID, NAME FROM STAFF WHERE JOB='Mgr' AND YEARS < 8
```

The result of the individual queries is listed in the section on UNION. The outcome of the two queries with INTERSECT is:

```
ID     NAME
------ ---------
   140 Fraye
   160 Molinare
```

When using the UNION, EXCEPT, and INTERSECT operators, keep the following in mind:

- All corresponding items in the select-lists of the queries for the operators must be compatible. See the data type compatibility table in the *SQL Reference* for more information.
- An ORDER BY clause, if used, must be placed after the last query with a set operator. The column name can only be used in the ORDER BY clause if the column has the same name for the corresponding items in the select list of the queries for every operator.
- Operations between columns that have the same data type and the same length produce a column with that type and length. See rules for result data types in the *SQL Reference* for the results of the UNION, EXCEPT, and INTERSECT set operators.

Predicates

Predicates let you construct conditions so that only those rows that meet these conditions are processed. Basic predicates are discussed in “Selecting Rows” on page 20. IN, BETWEEN, LIKE, EXISTS and quantified predicates are discussed in this section.

Using the IN Predicate

Use the IN predicate to compare a value with several other values. For example:

```
SELECT NAME
   FROM STAFF
   WHERE DEPT IN (20, 15)
```

This example is equivalent to:

```
SELECT NAME
   FROM STAFF
   WHERE DEPT = 20  OR DEPT = 15
```

You can use the IN and NOT IN operators when a subquery returns a set of values. For example, the following query lists the surnames of employees responsible for projects MA2100 and OP2012:

```
SELECT LASTNAME
   FROM EMPLOYEE
   WHERE EMPNO IN
      (SELECT RESPEMP
          FROM PROJECT
          WHERE PROJNO = 'MA2100'
          OR PROJNO = 'OP2012')
```

The subquery is evaluated once, and the resulting list is substituted directly into the outer-level query. For example, the subquery above selects employee numbers 10 and 330, the outer-level query is evaluated as if its WHERE clause were:

```
WHERE EMPNO IN (10, 330)
```

The list of values returned by the subquery can contain zero, one, or more values.

Using the BETWEEN Predicate

Use the BETWEEN predicate to compare a value with a range of values. The range is inclusive and it considers the two expressions in the BETWEEN predicate for the comparisons.

The following example finds the names of employees who earn between $10, 000 and $20, 000:

```
SELECT LASTNAME
   FROM EMPLOYEE
   WHERE SALARY BETWEEN 10000 AND 20000
```

This is equivalent to:

```
SELECT LASTNAME
   FROM EMPLOYEE
   WHERE SALARY >= 10000 AND SALARY <= 20000
```

The next example finds the names of employees who earn less than $10, 000 or more than $20, 000:

```
SELECT LASTNAME
   FROM EMPLOYEE
   WHERE SALARY NOT BETWEEN 10000 AND 20000
```

Using the LIKE Predicate

Use the LIKE predicate to search for strings that have certain patterns. The pattern is specified through percentage signs and underscores.

- The underscore character (_) represents any single character.
- The percent sign (%) represents a string of zero or more characters.
- Any other character represents itself.

The following example selects employee names that are seven letters long starting with the letter ’S’:

```
SELECT NAME
   FROM STAFF
   WHERE NAME LIKE 'S_ _ _ _ _ _'
```

The next example selects names of employees that do not start with the letter ’S’:

```
SELECT NAME
   FROM STAFF
   WHERE NAME  NOT LIKE 'S%'
```

Using the EXISTS Predicate

You can use a subquery to test for the *existence* of a row that satisfies some condition. In this case, the subquery is linked to the outer-level query by the predicate EXISTS or NOT EXISTS.

When you link a subquery to an outer query by an EXISTS predicate, the subquery does not return a value. Rather, the EXISTS predicate is true if the answer set of the subquery contains one or more rows, and false if it contains no rows.

The EXISTS predicate is often used with correlated subqueries. The example below lists the departments that currently have no entries in the PROJECT table:

```
SELECT DEPTNO, DEPTNAME
   FROM DEPARTMENT X
   WHERE NOT EXISTS
            (SELECT *
                FROM PROJECT
                WHERE DEPTNO = X.DEPTNO)
   ORDER BY DEPTNO
```

You may connect the EXISTS and NOT EXISTS predicates to other predicates by using AND and OR in the WHERE clause of the outer-level query.

Quantified Predicates

A quantified predicate compares a value with a collection of values. If a fullselect returns more than one value, you must modify the comparison operators in your predicate by attaching the suffix ALL, ANY, or SOME. These suffixes determine how the set of values returned is to be treated in the outer-level predicate. The > comparison operator is used as an example (the remarks below apply to the other operators as well):

expression > ALL (fullselect)
The predicate is true if the expression is greater than each individual value returned by the fullselect. If the fullselect returns no values, the predicate is true. The result is false if the specified relationship is false for at least one value. Note that the <>ALL quantified predicate is equivalent to the NOT IN predicate.

The following example uses a subquery and a > ALL comparison to find the name and profession of all employees who earn more than all managers:

```
SELECT LASTNAME, JOB
   FROM EMPLOYEE
   WHERE SALARY > ALL
   (SELECT SALARY
       FROM EMPLOYEE
       WHERE JOB='MANAGER')
```

expression > ANY (fullselect)
The predicate is true if the expression is greater than at least one of the values returned by the fullselect. If the fullselect returns no values, the predicate is false. Note that the =ANY quantified operator is equivalent to the IN predicate.

expression > SOME (fullselect)
SOME is synonymous with ANY.

For more information on predicates and operators, refer to the *SQL Reference*.

Chapter 7. Advanced SQL

This chapter covers several features of DB2 Universal Database that allow you to design queries more effectively, while customizing them to your needs. Topics in this chapter are based upon your thorough understanding of the material up to this point.

This chapter covers:

- Enforcing Business Rules with Constraints and Triggers
- Joins
- ROLLUP and CUBE Queries and Recursive Queries

Enforcing Business Rules with Constraints and Triggers

In the business world we quite often need to make sure certain rules are always enforced. For instance, an employee working on a project has to be on the payroll list. Or, we want certain events to happen systematically. For instance, if a salesperson makes a sale, their commission should be increased.

DB2 Universal Database offers a useful suite of methods to this end. *Unique constraints* is the rule that forbids duplicate values in one or more columns of a table. *Referential integerity constraints* ensure the data consistency across the specified tables. *Table check constraints* are conditions that are defined as part of the table definition that restrict the values used in one or more columns. *Triggers* allow you to define a set of actions that are executed, or triggered, by a delete, insert, or update operation on a specified table. Triggers can be used for writing to other tables, for modifying of input values, and for the issuing alert messages.

The first section provides a conceptual overview of keys. Later, referential integerity, constraints, and triggers are explored through examples and diagrams.

Keys

A *key* is a set of columns that you can use to identify or access a particular row or rows.

A key composed of more than one column is called a *composite key*. In a table with a composite key, the ordering of the columns within the composite key is not constrained by their ordering within the table.

Unique Keys

A *unique key* is defined to have no two of its values the same. The columns of a unique key cannot contain null values. The constraint is enforced by the database manager during the execution of INSERT and UPDATE statements. A table can have mulitple unique keys. Unique keys are optional and can be defined in CREATE TABLE or ALTER TABLE statements.

Primary Keys

A *primary key* is a unique key that is a part of the definition of the table. A table cannot have more than one primary key, and the columns of a primary key cannot contain null values. Primary keys are optional and can be defined in CREATE TABLE or ALTER TABLE statements.

Foreign Keys

A *foreign key* is specified in the definition of a referential constraint. A table can have zero or more foreign keys. The value of the composite foreign key is null if any component of the value is null. Foreign keys are optional and can be defined in CREATE TABLE statements or ALTER TABLE statements.

Unique Constraints

A unique constraint ensures that values of a key are unique within a table. Unique constraints are optional, and you can define them using the CREATE TABLE or ALTER TABLE statements by specifying the PRIMARY KEY or UNIQUE clause. For example, you can define a unique constraint on the employee number column of a table to ensure that every employee has a unique number.

Referential Integrity Constraints

By defining unique constraints and foreign keys you can define relationships between tables and consequently enforce certain business rules. The combination of unique key and foreign key constraints is commonly referred to as referential integrity constraints. A unique constraint referenced by a foreign key is called a *parent key*. A foreign key refers to or is related to a specific parent key. For example, a rule might state that every employee (EMPLOYEE table) must belong to an existing department (DEPARTMENT table). So, we define department number in the EMPLOYEE table as foreign key, and department number in the DEPARTMENT table as the primary key. The following diagram provides a visual description of referential integrity constraints.

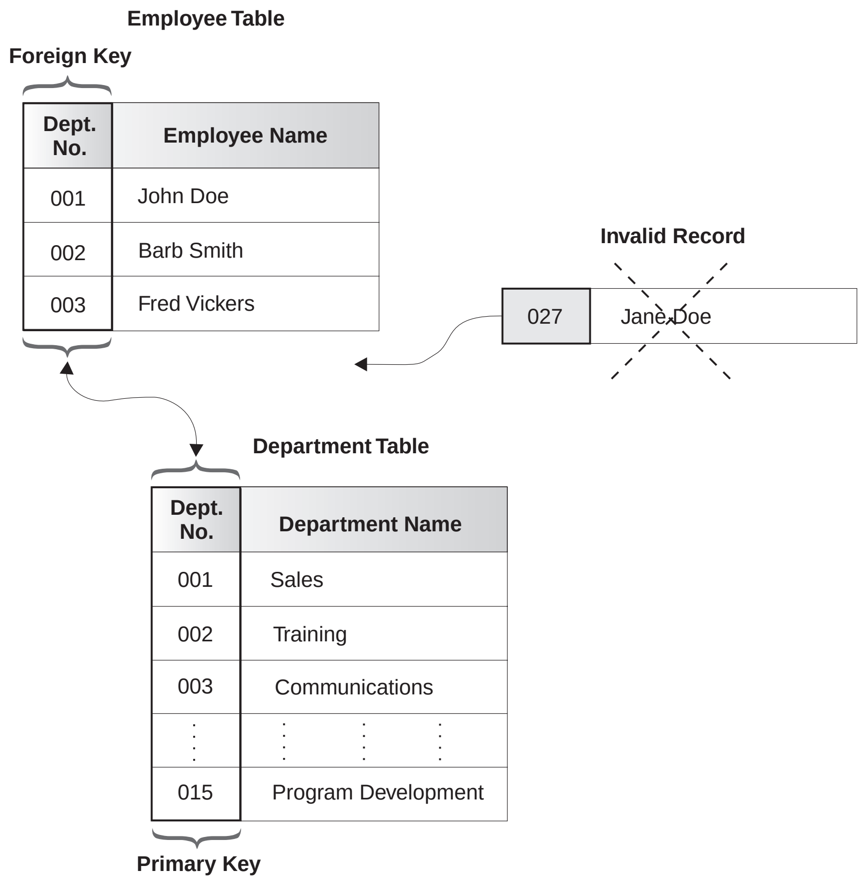


Figure 4. Foreign and Primary Constraints Define Relationships and Protect Data

Table Check Constraints

Table check constraints specify conditions that are evaluated for each row of a table. You can specify check constraints on individual columns. You can add them by using the CREATE or ALTER TABLE statements.

The following statement creates a table with the following constraints:

- The values of the department number must lie in the range 10 to 100
- The job of an employee can only be one of the following: “Sales”, “Mgr”, or “Clerk”
- Every employee who was hired prior to 1986 must make more than $40, 500.

```
CREATE TABLE EMP
      (ID          SMALLINT NOT NULL,
       NAME        VARCHAR(9),
       DEPT        SMALLINT CHECK (DEPT BETWEEN 10 AND 100),
       JOB         CHAR(5)   CHECK (JOB IN ('Sales', 'Mgr', 'Clerk')),
       HIREDATE    DATE,
       SALARY      DECIMAL(7,2),
       COMM        DECIMAL(7,2),
       PRIMARY KEY (ID),
       CONSTRAINT YEARSAL CHECK (YEAR(HIREDATE) >= 1986 OR SALARY > 40500) )
```

A constraint is violated only if the condition evaluates to false. For example, if DEPT is NULL for an inserted row, the insert proceeds without error, even though values for DEPT should be between 10 and 100 as defined in the constraint.

The following statement adds a constraint to the EMPLOYEE table named COMP that an employee's total compensation must exceed $15, 000:

```
ALTER TABLE EMP
   ADD CONSTRAINT COMP CHECK (SALARY + COMM > 15000)
```

The existing rows in the table will be checked to ensure that they do not violate the new constraint. You can defer this checking by using the SET CONSTRAINTS statement as follows:

```
SET CONSTRAINTS FOR EMP OFF
ALTER TABLE EMP ADD CONSTRAINT COMP CHECK (SALARY + COMM > 15000)
SET CONSTRAINTS FOR EMP IMMEDIATE CHECKED
```

First, the SET CONSTRAINTS statement is used to defer constraint checking for the table. Then one or more constraints can be added to the table without checking the constraints. Then the SET CONSTRAINTS statement is issued again to turn constraint checking back on and to perform any deferred constraint checking.

Triggers

A *trigger* defines a set of actions that is activated by an operation that modifies the data in a specified base table.

You can use triggers to perform validation of input data, to automatically generate a value for a newly inserted row, to read from other tables for cross-referencing purposes, to write to other tables for audit-trail purposes, or to support alerts through electronic mail messages. Using triggers results in faster application development, global enforcement of business rules, and easier maintenance of applications and data.

DB2 Universal Database supports several types of triggers. Triggers can be defined to be activated either before or after a DELETE, INSERT, or UPDATE operation. Each trigger includes a set of SQL statements called a *triggered action* that can include an optional search condition.

After triggers can be further defined to perform the triggered action either for each row or once for the statement, while *before triggers* always perform the triggered action for each row.

Use a trigger before an INSERT, UPDATE, or DELETE statement to check for certain conditions before performing a triggering operation or to change the input values before they are stored in the table. Use an after trigger to propagate values as necessary or perform other tasks, such as sending a message, that may be required as a part of the trigger operation.

The following example illustrates a use of before and after triggers. Consider an application that records and tracks changes to stock prices. The database contains two tables, CURRENTQUOTE and QUOTEHISTORY defined as:

```
CREATE TABLE CURRENTQUOTE
(SYMBOL VARCHAR(10),
 QUOTE DECIMAL(5,2),
 STATUS VARCHAR(9))

CREATE TABLE  QUOTEHISTORY
(SYMBOL VARCHAR(10),
 QUOTE DECIMAL(5,2),
 TIMESTAMP TIMESTAMP)
```

When the QUOTE column of CURRENTQUOTE is updated using a statement such as:

```
UPDATE CURRENTQUOTE
   SET QUOTE = 68.5
   WHERE SYMBOL = 'IBM'
```

The STATUS column of CURRENTQUOTE should be updated to reflect whether the stock is:

- Rising in value
- At a new high for the year
- Dropping in value
- At a new low for the year
- Steady in value.

This is done using the following before trigger:

1

```
CREATE TRIGGER STOCK_STATUS
   NO CASCADE BEFORE UPDATE OF QUOTE ON CURRENTQUOTE
   REFERENCING NEW AS NEWQUOTE OLD AS OLDQUOTE
   FOR EACH ROW MODE DB2SQL
```

2

```
 SET NEWQUOTE.STATUS =
```

3

```
    CASE
```

4

```
       WHEN NEWQUOTE.QUOTE >=
                 (SELECT MAX(QUOTE)
                     FROM QUOTEHISTORY
                     WHERE SYMBOL = NEWQUOTE.SYMBOL
                       AND YEAR(TIMESTAMP) = YEAR(CURRENT DATE) )
       THEN 'High'
```

5

```
     WHEN NEWQUOTE.QUOTE <=
                 (SELECT MIN(QUOTE)
                     FROM QUOTEHISTORY
                     WHERE SYMBOL = NEWQUOTE.SYMBOL
                     AND YEAR(TIMESTAMP) = YEAR(CURRENT DATE) )
     THEN 'Low'
```

6

```
     WHEN NEWQUOTE.QUOTE > OLDQUOTE.QUOTE
        THEN 'Rising'
     WHEN NEWQUOTE.QUOTE < OLDQUOTE.QUOTE
        THEN 'Dropping'
     WHEN NEWQUOTE.QUOTE = OLDQUOTE.QUOTE
        THEN 'Steady'
  END
```

1 This block of code defines a trigger named STOCK_STATUS as a trigger that should be activated before the update of the QUOTE column of the CURRENTQUOTE table. The second line specifies that the triggered action is to be applied before any changes caused by the actual update of the CURRENTQUOTE table are applied to the database. It also means that the triggered action will not cause any other triggers to be activated. The third line specifies the names that must be used as qualifiers of the column name for the new values (NEWQUOTE) and the old values (OLDQUOTE). Column names qualified with these correlation names (NEWQUOTE and OLDQUOTE) are called *transition variables*. The fourth line indicates that the triggered action should be executed for each row.

2 This marks the start of the first and only SQL statement in the triggered action of this trigger. The SET transition-variable statement is used in a trigger to assign a value to a column in the row of the table that is being updated by the statement that activated the trigger. This statement is assigning a value to the STATUS column of the CURRENTQUOTE table.

3 The expression that is used on the right hand side of the assignment is a CASE expression. The CASE expression extends to the END keyword.

4 The first case checks to see if the new quote (NEWQUOTE.QUOTE) exceeds the maximum value for the stock symbol in the current calendar year. The subquery is using the QUOTEHISTORY table that is updated by the after trigger that follows.

5 The second case checks to see if the new quote (NEWQUOTE.QUOTE) is less than the minimum value for the stock symbol in the current calendar year. The subquery is using the QUOTEHISTORY table that is updated by the after trigger that follows.

6 The last three cases compare the new quote (NEWQUOTE.QUOTE) to the quote that was in the table (OLDQUOTE.QUOTE) to determine if it is greater, less or the same. The SET transition-variable statement ends here.

In addition to updating the entry in the CURRENTQUOTE table, an audit record needs to be created by copying the new quote, with a timestamp, to the QUOTEHISTORY table. This is done using the following after trigger:

1
```
CREATE TRIGGER RECORD_HISTORY
AFTER UPDATE OF QUOTE ON CURRENTQUOTE
REFERENCING NEW AS NEWQUOTE
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
```

2
```
INSERT INTO QUOTEHISTORY
   VALUES (NEWQUOTE.SYMBOL, NEWQUOTE.QUOTE, CURRENT TIMESTAMP);
END
```

1 This block of code defines a trigger named RECORD_HISTORY as a trigger that should be activated after the update of the QUOTE column of the CURRENTQUOTE table. The third line specifies the name that should be used as a qualifier of the column name for the new value (NEWQUOTE). The fourth line indicates that the triggered action should be executed for each row.

2 The triggered action of this trigger includes a single SQL statement that inserts a row into the QUOTEHISTORY table using the data from the row that has been updated (NEWQUOTE.SYMBOL and NEWQUOTE.QUOTE) and the current timestamp.

CURRENT TIMESTAMP is a special register containing the timestamp. A list and explanation is provided in “Special Registers” on page 68.

Joins

The process of combining data from two or more tables is called joining tables. The database manager forms all combinations of rows from the specified tables. For each combination, it tests the *join condition*. A join condition is a search condition, with some restrictions. For a list of restrictions refer to the *SQL Reference*.

Note that the data types of the columns involved in the join condition do not have to be identical; however, they must be compatible. The join condition is evaluated the same way as any other search condition, and the same rules for comparisons apply.

If you do not specify a join condition, all combinations of rows from tables listed in the FROM clause are returned, even though the rows may be completely unrelated. The result is referred to as the *cross product* of the two tables.

Examples in this section are based on the next two tables. They are simplifications of the tables from the sample database but do not exist in the sample database. They are used to outline interesting points about joins in general. SAMP_STAFF lists the name of employees who are not employed as contractors and their job descriptions, while SAMP_PROJECT lists the name of employees (contract and full-time) and the projects that they are working on.

The tables are as follows:

NAME	PROJ
Haas	AD3100
Thompson	PL2100
Walker	MA2112
Lutz	MA2111

Figure 5. SAMP_PROJECT TABLE

NAME	JOB
Haas	PRES
Thompson	MANAGER
Lucchessi	SALESREP
Nicholls	ANALYST

Figure 6. SAMP_STAFF TABLE

The following example produces the cross product of two table. A join condition is not specified, so all combination of rows is present:

```
     SELECT SAMP_PROJECT.NAME,
            SAMP_PROJECT.PROJ, SAMP_STAFF.NAME, SAMP_STAFF.JOB
        FROM SAMP_PROJECT, SAMP_STAFF
```

This statement produces the following result:

```
     NAME       PROJ   NAME       JOB
     ---------- ------ ---------- --------
     Haas       AD3100 Haas       PRES
     Thompson   PL2100 Haas       PRES
     Walker     MA2112 Haas       PRES
     Lutz       MA2111 Haas       PRES
     Haas       AD3100 Thompson   MANAGER
     Thompson   PL2100 Thompson   MANAGER
     Walker     MA2112 Thompson   MANAGER
     Lutz       MA2111 Thompson   MANAGER
     Haas       AD3100 Lucchessi  SALESREP
     Thompson   PL2100 Lucchessi  SALESREP
     Walker     MA2112 Lucchessi  SALESREP
     Lutz       MA2111 Lucchessi  SALESREP
     Haas       AD3100 Nicholls   ANALYST
     Thompson   PL2100 Nicholls   ANALYST
     Walker     MA2112 Nicholls   ANALYST
     Lutz       MA2111 Nicholls   ANALYST
```

The two main types of joins are *inner joins* and *outer joins*. So far, in all of our examples we have used the inner join. Inner joins keep only the rows from the cross product that meet the join condition. If a row exists in one table, but not the other, the information is not included in the result table.

The following example produces the inner join of the two tables. The inner join lists the information full-time employees who are assigned to a project :

```
SELECT SAMP_PROJECT.NAME,
       SAMP_PROJECT.PROJ, SAMP_STAFF.NAME, SAMP_STAFF.JOB
   FROM SAMP_PROJECT, SAMP_STAFF
   WHERE SAMP_STAFF.NAME = SAMP_PROJECT.NAME
```

Alternately, you can specify the inner join as follows:

```
SELECT SAMP_PROJECT.NAME,
       SAMP_PROJECT.PROJ, SAMP_STAFF.NAME, SAMP_STAFF.JOB
   FROM SAMP_PROJECT INNER JOIN SAMP_STAFF
     ON SAMP_STAFF.NAME = SAMP_PROJECT.NAME
```

The result is:

```
NAME       PROJ   NAME       JOB
---------- ------ ---------- --------
Haas       AD3100 Haas       PRES
Thompson   PL2100 Thompson   MANAGER
```

Note that the result of the inner join consists of rows that have matching values for the NAME column in the right and the left tables - both 'Haas' and 'Thompson' are included in the SAMP_STAFF table that lists all full-time employee and in the SAMP_PROJECT table that lists full-time and contract employee assigned to a project.

Outer joins are a concatentation of the inner join and rows from the left table, right table, or both tables that are missing from the inner join. When you perform an outer join on two tables, you arbitrarily assign one table as the left table and the other one as the right table. There are three types of outer joins:

1. ***left outer join*** includes the inner join and the rows from the left table that are not included in the inner join.
2. ***right outer join*** includes the inner join and the rows from the right table that are not included in the inner join.
3. ***full outer join*** includes the inner join and the rows from both the left and right tables that are not included in the inner join.

Use the SELECT statement to specify the columns to be displayed. In the FROM clause, list the name of the first table followed by the keywords LEFT OUTER JOIN, RIGHT OUTER JOIN or FULL OUTER JOIN. Next you need to specify the second table followed by the ON keyword. Following the ON keyword, specify the join condition to express a relationship between the tables to be joined.

In the following example, SAMP_STAFF is designated as the right table and SAMP_PROJECT as the left table. By using LEFT OUTER JOIN, we list the name and project number of all employees, full-time and contract, (listed in SAMP_PROJECT) and their job title, if they are a full-time employee (listed in SAMP_STAFF):

```
SELECT SAMP_PROJECT.NAME, SAMP_PROJECT.PROJ,
       SAMP_STAFF.NAME, SAMP_STAFF.JOB
   FROM SAMP_PROJECT LEFT OUTER JOIN SAMP_STAFF
     ON SAMP_STAFF.NAME = SAMP_PROJECT.NAME
```

This statement produces the following result:

```
NAME       PROJ                 NAME       JOB
---------- -------------------- ---------- --------------------
Haas       AD3100               Haas       PRES
Lutz       MA2111               -          -
Thompson   PL2100               Thompson   MANAGER
Walker     MA2112               -          -
```

Rows with values in all columns are the result of the inner join. These are rows that satisfy the join condition: ’Haas’ and ’Thompson’ are listed in both SAMP_PROJECT (left table) and SAMP_STAFF (right table). For rows that the join condition was not satisfied, the null value appears on columns of the right table: ’Lutz’ and ’Walker’ are contract employees listed in the SAMP_PROJECT table and not in the SAMP_STAFF table. Note that all rows from the left table are included in the result set.

In the next example, SAMP_STAFF is designated as the right table and SAMP_PROJECT as the left table. By using RIGHT OUTER JOIN we list the name and job title of all full-time employees (listed in SAMP_STAFF) and their project number, if they are assigned to one (listed in SAMP_PROJECT):

```
SELECT SAMP_PROJECT.NAME,
       SAMP_PROJECT.PROJ, SAMP_STAFF.NAME, SAMP_STAFF.JOB
   FROM SAMP_PROJECT RIGHT OUTER JOIN SAMP_STAFF
     ON SAMP_STAFF.NAME = SAMP_PROJECT.NAME
```

The result is:

```
NAME       PROJ                 NAME       JOB
---------- -------------------- ---------- --------------------
Haas       AD3100               Haas       PRES
-          -                    Lucchessi  SALESREP
-          -                    Nicholls   ANALYST
Thompson   PL2100               Thompson   MANAGER
```

As in the left outer join, rows with values in all columns are the result of the inner join. These are rows that satisfy the join condition: ’Haas’ and ’Thompson’ are listed in both SAMP_PROJECT (left table) and SAMP_STAFF (right table). For rows that the join condition was not satisfied, the null value appears on columns of the right table: ’Lucchessi’ and ’Nicholls’ are full-time employee that are not assigned to a project. While they are listed in SAMP_STAFF, they are not in SAMP_PROJECT. Note that all rows from the right table are included in the result set.

The next example uses FULL OUTER JOIN with the SAMP_PROJECT and SAMP_STAFF tables. It lists the name of all full-time, including the ones that are not assigned to a project, and contract employees:

```
SELECT SAMP_PROJECT.NAME, SAMP_PROJECT.PROJ,
       SAMP_STAFF.NAME, SAMP_STAFF.JOB
   FROM SAMP_PROJECT FULL OUTER JOIN SAMP_STAFF
     ON SAMP_STAFF.NAME = SAMP_PROJECT.NAME
```

The result is:

```
NAME       PROJ                 NAME       JOB
---------- -------------------- ---------- --------------------
Haas       AD3100               Haas       PRES
-          -                    Lucchessi  SALESREP
-          -                    Nicholls   ANALYST
Thompson   PL2100               Thompson   MANAGER
Lutz       MA2111               -          -
Walker     MA2112               -          -
```

This result includes the left outer join, the right outer join and the inner join. All full-time and contract employees are listed. Just like left outer join and right outer join, for values that the join condition was not satisfied the null value appears in the respective column. Every row from SAMP_STAFF and SAMP_PROJECT is included in the result set.

Complex Queries

DB2 Universal Database allows you to group, consolidate, and view multiple columns in a single result set through the use of ROLLUP and CUBE. This new and powerful capability enhances and simplifies SQL based data analysis.

There are various methods of extracting useful information from the database. You can implement recursive queries to produce result tables from existing data sets.

ROLLUP and CUBE Queries

You specify ROLLUP and CUBE operations in the GROUP BY clause of a query. ROLLUP grouping produces a result set containing the regular grouped rows and *sub-total rows*. CUBE grouping produces a result set containing the rows from ROLLUP and cross-tabulation rows. So for ROLLUP, you can get the sales by person by month with monthly sales totals and an overall total. For CUBE, additional rows would be included for total sales by person. See the *SQL Reference* for further details.

Recursive Queries

A *recursive query* is a query that iteratively uses result data to determine further results. You might think of this as traversing a tree or a graph. Practical examples where this is useful include bill of materials applications, reservation systems, network planning and scheduling. A recursive query is written using a common table expression that includes a reference to its own name. See the *SQL Reference* for examples of recursive queries.

Chapter 8. Customizing and Enhancing Data Manipulation

This chapter gives a brief introduction to *object-oriented extensions* in DB2 Universal Database. There are many advantages to using object oriented extensions. *User-defined Types (UDT)* increase the set of data types available to your applications while *user-defined Functions (UDF)* allow for creation of application specific functions. UDFs act as *methods* for UDTs by providing consistent behavior and encapsulation of the types.

Special registers and *system catalogs* are discussed next. Special registers provide information about the connection. The system catalogs contain information about the logical and the physical structure of database objects.

This chapter covers:

- User-Defined Types
- User-Defined Functions
- Large Objects (LOBs)
- Special Registers
- Introduction to Catalog Views

A detailed discussion of the above topics is beyond the scope of this book but is presented in the *SQL Reference* and *Administration Guide*.

User-Defined Types

A *distinct type* is a user-defined data type that shares its internal representation with an existing type (its “source” type), but is considered to be separate and incompatible for most operations. For example, you might want to define an age type, a weight type, and a height type, all of which have quite different semantics, but which use the built-in data type INTEGER for their internal representations.

The following example illustrates the creation of a distinct type named PAY:

```
CREATE DISTINCT TYPE PAY AS DECIMAL(9,2) WITH COMPARISONS
```

Although PAY has the same representation as the built-in data type DECIMAL(9,2), it is considered to be a separate type that is not comparable to DECIMAL(9,2) or to any other type. It is comparable only to the same distinct type. Also, operators and functions that would work on DECIMAL will not apply here. For example, a value with PAY data type cannot be multiplied with a value of INTEGER data type. Therefore, you have to write functions that only apply to the PAY data type.

Using distinct data types limits accidental mistakes. For instance, if the SALARY column of table EMPLOYEE was defined as a PAY data type, it could not be added to COMM even though their sourced types are the same.

Distinct data types support casting. A source type can be cast to a distinct data type, and a distinct data type to a source type. For example, if the SALARY column of the table EMPLOYEE were defined as a PAY data type, the following example would not fail at the comparison operator.

```
SELECT * FROM EMPLOYEE
   WHERE DECIMAL(SALARY) = 41250
```

DECIMAL(SALARY) returns a decimal data type. Inversely, a numeric data type can be cast to a PAY type. For example, you can cast the number 41250 by using PAY(41250).

User-Defined Functions

As mentioned in “Using Functions” on page 28, DB2 Universal Database provides built-in and user-defined functions (UDF). However, this set of functions will never satisfy all requirements. Often, you need to create customized functions for particular tasks. User-defined functions allow you to create customized functions.

There are two types of user-defined functions: *sourced* and *external*.

Sourced user-defined functions allow for user-defined types to selectively reference another built-in or user-defined function that is already known to the database. You can use both scalar and column functions.

In the next example a user-defined function called MAX is created that is based on the built-in MAX column function, which takes a DECIMAL data type as input. The MAX UDF takes a PAY type as input and returns a PAY type as output.

```
CREATE FUNCTION MAX(PAY) RETURNS PAY
   SOURCE MAX(DECIMAL)
```

External user-defined functions are written by users in a programming language. There are *external scalar functions* and *external table functions* and both are discussed in the *SQL Reference*.

Assuming that you have already written a function that counts the number of words in a string, you can register it with the database using the CREATE FUNCTION statement with the name WORDCOUNT. This function can then be used in SQL statements.

For example, the following statement returns employee numbers and the number of words in the ASCII form of their resumes. WORDCOUNT is an external scalar function that has been registered with the database by the user and is now being used in the statement.

```
SELECT EMPNO, WORDCOUNT(RESUME)
   FROM EMP_RESUME
   WHERE RESUME_FORMAT = 'ascii'
```

For more detailed information on writing user-defined functions, refer to the *Application Development Guide*.

Large Objects (LOBs)

The term *large object* and its acronym *LOB* are used to refer to three data types: BLOB, CLOB, or DBCLOB. These types can contain large amounts of data, for objects such as audio, photos and documents.

A *Binary Large OBject (BLOB)* is a varying-length string, measured in bytes, that can be up to 2 gigabytes long. A BLOB is primarily intended to hold nontraditional data such as pictures, voice, and mixed media.

A *Character Large OBject (CLOB)* is a varying-length string, measured in bytes, that can be up to 2 gigabytes long. A CLOB is used to store large single-byte character set data such as documents. A CLOB is considered to be a character string.

A *Double-Byte Character Large OBject (DBCLOB)* is a varying-length string of double-byte characters that can be up to 2 gigabytes long (1 073 741 823 double-byte characters). A DBCLOB is used to store large double-byte character set data such as documents. A DBCLOB is considered to be a graphic string.

Manipulating Large Objects (LOBs)

Since LOB values can be very large, transferring them from the database server to client application program can be time consuming. However, typically LOB values are processed one piece at a time, rather than as a whole. For those cases where an application does not need (or want) the entire LOB value to be stored in application memory, it can reference this value via a *large object locator* variable.

Subsequent statements can then use the locators to perform operations on the data without necessarily retrieving the entire large object. Locator variables are used to reduce the storage requirements for the applications, and improve the performance by reducing the flow of data between the client and the server.

Another mechanism is *file reference variables*. They are used to retrieve a large object directly to a file or to update a large object in a table directly from a file. File reference variables are used to reduce the storage requirements for the applications since they do not need to store the large object data. For more information refer to the *Application Development Guide* and the *SQL Reference*.

Special Registers

A *special register* is a storage area that is defined for a connection by the database manager and is used to store information that can be referenced in SQL statements. Following are a few examples of the more commonly used special registers. For a list of all the special registers and more detailed information refer to the *SQL Reference*.

- CURRENT DATE: Holds the date according to the time-of-day clock at SQL statement execution time.
- CURRENT FUNCTION PATH: Holds a value that specifies the function path used to resolve function and data type references.
- CURRENT SERVER: Specifies the current application server.
- CURRENT TIME: Holds the time according to the time-of-day clock at the SQL statement execution time.
- CURRENT TIMESTAMP: Specifies a timestamp according to the time-of-day clock at SQL statement execution time.
- CURRENT TIMEZONE: Specifies the difference between Coordinated Universal Time and local time at the application server.
- USER: Specifies the run-time authorization ID.

You can display the contents of a special register with the VALUES statement. For example:

```
VALUES (CURRENT TIMESTAMP)
```

You could also use:

```
SELECT CURRENT TIMESTAMP FROM ORG
```

and this will return the TIMESTAMP for every row entry in the table.

Introduction to Catalog Views

DB2 creates and maintains an extensive set of system catalog tables for each database. These tables contain information about the logical and physical structure of database objects such as tables, views, packages, referential integrity relationships, functions, distinct types, and triggers. They are created when the database is created, and are updated in the course of normal operation. You cannot explicitly create or drop them, but you can query and view their contents.

For more information, refer to the *SQL Reference*.

Selecting Rows from System Catalogs

The catalog views are like any other database view. You can use SQL statements to look at the data, exactly in the same way that you would for any other view in the system.

You can find very useful information about tables in the SYSCAT.TABLES catalog. To find the names of existing tables that you have created, issue a statement similar to the following:

```
SELECT TABNAME, TYPE, CREATE_TIME
   FROM SYSCAT.TABLES
   WHERE DEFINER = USER
```

This statement produces the following result:

```
TABNAME            TYPE CREATE_TIME
------------------ ---- --------------------------
ORG                T    1997-05-22-11.15.27.850000
STAFF              T    1997-05-22-11.15.29.470000
DEPARTMENT         T    1997-05-22-11.15.30.850000
EMPLOYEE           T    1997-05-22-11.15.31.310000
EMP_ACT            T    1997-05-22-11.15.32.850000
PROJECT            T    1997-05-22-11.15.34.410007
EMP_PHOTO          T    1997-05-22-11.15.35.190000
EMP_RESUME         T    1997-05-22-11.15.40.600000
SALES              T    1997-05-22-11.15.43.000000
```

The following list includes catalog views pertaining to subjects discussed in this book. There are many other catalog views, and they are listed in detail in the *SQL Reference* and *Administration Guide*.

Description	Catalog View
check constraints	SYSCAT.CHECKS
columns	SYSCAT.COLUMNS
columns referenced by check constraints	SYSCAT.COLCHECKS

Description	Catalog View
columns used in keys	SYSCAT.KEYCOLUSE
datatypes	SYSCAT.DATATYPES
function parameters or result of a function	SYSCAT.FUNCPARMS
referential constraints	SYSCAT.REFERENCES
schemas	SYSCAT.SCHEMATA
table constraints	SYSCAT.TABCONST
tables	SYSCAT.TABLES
triggers	SYSCAT.TRIGGERS
user-defined functions	SYSCAT.FUNCTIONS
views	SYSCAT.VIEWS

Appendix A. Sample Tables

This appendix shows the information contained in the sample tables, and how to install and remove them. The sample tables are used in the examples that appear in this manual and other manuals in this library. In addition, the data contained in the sample files with BLOB and CLOB data types is shown.

The following sections are included in this appendix:.

"Walker Resume" on page 88.

In the sample tables, a dash (-) indicates a null value.

The Sample Database

The examples in this book use a sample database. To use these examples, you must install the SAMPLE database. To use it, the database manager must be installed.

To Install the Sample Database

An executable file installs the sample database.[2] To install a database you must have SYSADM authority.

- **When Using UNIX-based Systems**

  If you are using the operating system command prompt, type:

  ```
  sqllib/misc/db2sampl <path>
  ```

  from the home directory of the database manager instance owner, where *path* is an optional parameter specifying the path where the sample database is to be created. Press Enter.[3] The schema for DB2SAMPL is the CURRENT SCHEMA special register value.
- **When using OS/2, Windows 95 or Windows NT**

  If you are using the operating system command prompt, type:

  ```
  db2sampl e
  ```

  where *e* is an optional parameter specifying the drive where the database is to be created. Press Enter.[4]

  If you are not logged on to your workstation through User Profile Management, you will be prompted to do so.

To Erase the Sample Database

If you do not need to access the sample database, you can erase it by using the DROP DATABASE command:

```
db2 drop database sample
```

2. For information related to this command, see the DB2SAMPL command in the *Command Reference*.
3. If the path parameter is not specified, the sample tables are installed in the default path specified by the DFTDBPATH parameter in the database manager configuration file.
4. If the drive parameter is not specified, the sample tables are installed on the same drive as DB2.

CL_SCHED Table

Name:	**CLASS_CODE**	**DAY**	**STARTING**	**ENDING**
Type:	char(7)	smallint	time	time
Desc:	Class Code (room:teacher)	Day # of 4 day schedule	Class Start Time	Class End Time

DEPARTMENT Table

Name:	**DEPTNO**	**DEPTNAME**	**MGRNO**	**ADMRDEPT**	**LOCATION**
Type:	char(3) not null	varchar(29) not null	char(6)	char(3) not null	char(16)
Desc:	Department number	Name describing general activities of department	Employee number (EMPNO) of department manager	Department (DEPTNO) to which this department reports	Name of the remote location
Values:	A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	-
	B01	PLANNING	000020	A00	-
	C01	INFORMATION CENTER	000030	A00	-
	D01	DEVELOPMENT CENTER	-	A00	-
	D11	MANUFACTURING SYSTEMS	000060	D01	-
	D21	ADMINISTRATION SYSTEMS	000070	D01	-
	E01	SUPPORT SERVICES	000050	A00	-
	E11	OPERATIONS	000090	E01	-
	E21	SOFTWARE SUPPORT	000100	E01	-

EMPLOYEE Table

Names:	**EMPNO**	**FIRSTNME**	**MIDINIT**	**LASTNAME**	**WORKDEPT**	**PHONENO**	**HIREDATE**
Type:	char(6) not null	varchar(12) not null	char(1) not null	varchar(15) not null	char(3)	char(4)	date
Desc:	Employee number	First name	Middle initial	Last name	Department (DEPTNO) in which the employee works	Phone number	Date of hire

JOB	**EDLEVEL**	**SEX**	**BIRTHDATE**	**SALARY**	**BONUS**	**COMM**
char(8)	smallint not null	char(1)	date	dec(9,2)	dec(9,2)	dec(9,2)
Job	Number of years of formal education	Sex (M male, F female)	Date of birth	Yearly salary	Yearly bonus	Yearly commission

See the following page for the values in the EMPLOYEE table.

EMPNO	FIRSTNME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	ED LEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
char(6) not null	varchar(12) not null	char(1) not null	varchar(15) not null	char(3)	char(4)	date	char(8)	smallint not null	char(1)	date	dec(9,2)	dec(9,2)	dec(9,2)
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN		O'CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FIELDREP	16	M	1932-08-11	19950	400	1596

EMPNO	FIRSTNME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	ED LEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
000330	WING		LEE	E21	2103	1976-02-23	FIELDREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FIELDREP	16	M	1926-05-17	23840	500	1907

EMP_ACT Table

Name:	**EMPNO**	**PROJNO**	**ACTNO**	**EMPTIME**	**EMSTDATE**	**EMENDATE**
Type:	char(6) not null	char(6) not null	smallint not null	dec(5,2)	date	date
Desc:	Employee number	Project number	Activity number	Proportion of employee’s time spent on project	Date activity starts	Date activity ends
Values:	000010	AD3100	10	.50	1982-01-01	1982-07-01
	000070	AD3110	10	1.00	1982-01-01	1983-02-01
	000230	AD3111	60	1.00	1982-01-01	1982-03-15
	000230	AD3111	60	.50	1982-03-15	1982-04-15
	000230	AD3111	70	.50	1982-03-15	1982-10-15
	000230	AD3111	80	.50	1982-04-15	1982-10-15
	000230	AD3111	180	1.00	1982-10-15	1983-01-01
	000240	AD3111	70	1.00	1982-02-15	1982-09-15
	000240	AD3111	80	1.00	1982-09-15	1983-01-01
	000250	AD3112	60	1.00	1982-01-01	1982-02-01
	000250	AD3112	60	.50	1982-02-01	1982-03-15
	000250	AD3112	60	.50	1982-12-01	1983-01-01
	000250	AD3112	60	1.00	1983-01-01	1983-02-01
	000250	AD3112	70	.50	1982-02-01	1982-03-15
	000250	AD3112	70	1.00	1982-03-15	1982-08-15
	000250	AD3112	70	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.50	1982-10-15	1982-12-01

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	000250	AD3112	180	.50	1982-08-15	1983-01-01
	000260	AD3113	70	.50	1982-06-15	1982-07-01
	000260	AD3113	70	1.00	1982-07-01	1983-02-01
	000260	AD3113	80	1.00	1982-01-01	1982-03-01
	000260	AD3113	80	.50	1982-03-01	1982-04-15
	000260	AD3113	180	.50	1982-03-01	1982-04-15
	000260	AD3113	180	1.00	1982-04-15	1982-06-01
	000260	AD3113	180	.50	1982-06-01	1982-07-01
	000270	AD3113	60	.50	1982-03-01	1982-04-01
	000270	AD3113	60	1.00	1982-04-01	1982-09-01
	000270	AD3113	60	.25	1982-09-01	1982-10-15
	000270	AD3113	70	.75	1982-09-01	1982-10-15
	000270	AD3113	70	1.00	1982-10-15	1983-02-01
	000270	AD3113	80	1.00	1982-01-01	1982-03-01
	000270	AD3113	80	.50	1982-03-01	1982-04-01
	000030	IF1000	10	.50	1982-06-01	1983-01-01
	000130	IF1000	90	1.00	1982-01-01	1982-10-01
	000130	IF1000	100	.50	1982-10-01	1983-01-01
	000140	IF1000	90	.50	1982-10-01	1983-01-01
	000030	IF2000	10	.50	1982-01-01	1983-01-01
	000140	IF2000	100	1.00	1982-01-01	1982-03-01
	000140	IF2000	100	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-10-01	1983-01-01
	000010	MA2100	10	.50	1982-01-01	1982-11-01
	000110	MA2100	20	1.00	1982-01-01	1982-03-01
	000010	MA2110	10	1.00	1982-01-01	1983-02-01
	000200	MA2111	50	1.00	1982-01-01	1982-06-15
	000200	MA2111	60	1.00	1982-06-15	1983-02-01
	000220	MA2111	40	1.00	1982-01-01	1983-02-01
	000150	MA2112	60	1.00	1982-01-01	1982-07-15
	000150	MA2112	180	1.00	1982-07-15	1983-02-01
	000170	MA2112	60	1.00	1982-01-01	1983-06-01
	000170	MA2112	70	1.00	1982-06-01	1983-02-01
	000190	MA2112	70	1.00	1982-02-01	1982-10-01
	000190	MA2112	80	1.00	1982-10-01	1983-10-01
	000160	MA2113	60	1.00	1982-07-15	1983-02-01

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	000170	MA2113	80	1.00	1982-01-01	1983-02-01
	000180	MA2113	70	1.00	1982-04-01	1982-06-15
	000210	MA2113	80	.50	1982-10-01	1983-02-01
	000210	MA2113	180	.50	1982-10-01	1983-02-01
	000050	OP1000	10	.25	1982-01-01	1983-02-01
	000090	OP1010	10	1.00	1982-01-01	1983-02-01
	000280	OP1010	130	1.00	1982-01-01	1983-02-01
	000290	OP1010	130	1.00	1982-01-01	1983-02-01
	000300	OP1010	130	1.00	1982-01-01	1983-02-01
	000310	OP1010	130	1.00	1982-01-01	1983-02-01
	000050	OP2010	10	.75	1982-01-01	1983-02-01
	000100	OP2010	10	1.00	1982-01-01	1983-02-01
	000320	OP2011	140	.75	1982-01-01	1983-02-01
	000320	OP2011	150	.25	1982-01-01	1983-02-01
	000330	OP2012	140	.25	1982-01-01	1983-02-01
	000330	OP2012	160	.75	1982-01-01	1983-02-01
	000340	OP2013	140	.50	1982-01-01	1983-02-01
	000340	OP2013	170	.50	1982-01-01	1983-02-01
	000020	PL2100	30	1.00	1982-01-01	1982-09-15

EMP_PHOTO Table

Name:	EMPNO	PHOTO_FORMAT	PICTURE
Type:	char(6) not null	varchar(10) not null	blob(100k)
Desc:	Employee number	Photo format	Photo of employee
Values:	000130	bitmap	db200130.bmp
	000130	gif	db200130.gif
	000130	xwd	db200130.xwd
	000140	bitmap	db200140.bmp
	000140	gif	db200140.gif
	000140	xwd	db200140.xwd
	000150	bitmap	db200150.bmp
	000150	gif	db200150.gif
	000150	xwd	db200150.xwd
	000190	bitmap	db200190.bmp
	000190	gif	db200190.gif
	000190	xwd	db200190.xwd

- “Quintana Photo” on page 83 shows the picture of the employee, Delores Quintana.
- “Nicholls Photo” on page 85 shows the picture of the employee, Heather Nicholls.
- “Adamson Photo” on page 87 shows the picture of the employee, Bruce Adamson.
- “Walker Photo” on page 88 shows the picture of the employee, James Walker.

EMP_RESUME Table

Name:	**EMPNO**	**RESUME_FORMAT**	**RESUME**
Type:	char(6) not null	varchar(10) not null	clob(5k)
Desc:	Employee number	Resume Format	Resume of employee
Values:	000130	ascii	db200130.asc
	000130	script	db200130.scr
	000140	ascii	db200140.asc
	000140	script	db200140.scr
	000150	ascii	db200150.asc
	000150	script	db200150.scr
	000190	ascii	db200190.asc
	000190	script	db200190.scr

- “Quintana Resume” on page 84 shows the resume of the employee, Delores Quintana.
- “Nicholls Resume” on page 85 shows the resume of the employee, Heather Nicholls.
- “Adamson Resume” on page 87 shows the resume of the employee, Bruce Adamson.
- “Walker Resume” on page 88 shows the resume of the employee, James Walker.

IN_TRAY Table

Name:	**RECEIVED**	**SOURCE**	**SUBJECT**	**NOTE_TEXT**
Type:	timestamp	char(8)	char(64)	varchar(3000)
Desc:	Date and Time received	User id of person sending note	Brief description	The note

ORG Table

Name:	**DEPTNUMB**	**DEPTNAME**	**MANAGER**	**DIVISION**	**LOCATION**
Type:	smallint not null	varchar(14)	smallint	varchar(10)	varchar(13)
Desc:	Department number	Department name	Manager number	Division of corporation	City
Values:	10	Head Office	160	Corporate	New York
	15	New England	50	Eastern	Boston
	20	Mid Atlantic	10	Eastern	Washington
	38	South Atlantic	30	Eastern	Atlanta
	42	Great Lakes	100	Midwest	Chicago
	51	Plains	140	Midwest	Dallas
	66	Pacific	270	Western	San Francisco
	84	Mountain	290	Western	Denver

PROJECT Table

Name:	**PROJNO**	**PROJNAME**	**DEPTNO**	**RESPEMP**	**PRSTAFF**	**PRSTDATE**	**PRENDATE**	**MAJPROJ**
Type:	char(6) not null	varchar(24) not null	char(3) not null	char(6) not null	dec(5,2)	date	date	char(6)
Desc:	Project number	Project name	Department responsible	Employee responsible	Estimated mean staffing	Estimated start date	Estimated end date	Major project, for a subproject
Values:	AD3100	ADMIN SERVICES	D01	000010	6.5	1982-01-01	1983-02-01	-
	AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	1982-01-01	1983-02-01	AD3100
	AD3111	PAYROLL PROGRAMMING	D21	000230	2	1982-01-01	1983-02-01	AD3110
	AD3112	PERSONNEL PROGRAMMING	D21	000250	1	1982-01-01	1983-02-01	AD3110
	AD3113	ACCOUNT PROGRAMMING	D21	000270	2	1982-01-01	1983-02-01	AD3110
	IF1000	QUERY SERVICES	C01	000030	2	1982-01-01	1983-02-01	-
	IF2000	USER EDUCATION	C01	000030	1	1982-01-01	1983-02-01	-
	MA2100	WELD LINE AUTOMATION	D01	000010	12	1982-01-01	1983-02-01	-
	MA2110	W L PROGRAMMING	D11	000060	9	1982-01-01	1983-02-01	MA2100
	MA2111	W L PROGRAM DESIGN	D11	000220	2	1982-01-01	1982-12-01	MA2110
	MA2112	W L ROBOT DESIGN	D11	000150	3	1982-01-01	1982-12-01	MA2110

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
	MA2113	W L PROD CONT PROGS	D11	000160	3	1982-02-15	1982-12-01	MA2110
	OP1000	OPERATION SUPPORT	E01	000050	6	1982-01-01	1983-02-01	-
	OP1010	OPERATION	E11	000090	5	1982-01-01	1983-02-01	OP1000
	OP2000	GEN SYSTEMS SERVICES	E01	000050	5	1982-01-01	1983-02-01	-
	OP2010	SYSTEMS SUPPORT	E21	000100	4	1982-01-01	1983-02-01	OP2000
	OP2011	SCP SYSTEMS SUPPORT	E21	000320	1	1982-01-01	1983-02-01	OP2010
	OP2012	APPLICATIONS SUPPORT	E21	000330	1	1982-01-01	1983-02-01	OP2010
	OP2013	DB/DC SUPPORT	E21	000340	1	1982-01-01	1983-02-01	OP2010
	PL2100	WELD LINE PLANNING	B01	000020	1	1982-01-01	1982-09-15	MA2100

SALES Table

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
Type:	date	varchar(15)	varchar(15)	int
Desc:	Date of sales	Employee’s last name	Region of sales	Number of sales
Values:	12/31/1995	LUCCHESSI	Ontario-South	1
	12/31/1995	LEE	Ontario-South	3
	12/31/1995	LEE	Quebec	1
	12/31/1995	LEE	Manitoba	2
	12/31/1995	GOUNOT	Quebec	1
	03/29/1996	LUCCHESSI	Ontario-South	3
	03/29/1996	LUCCHESSI	Quebec	1
	03/29/1996	LEE	Ontario-South	2
	03/29/1996	LEE	Ontario-North	2
	03/29/1996	LEE	Quebec	3
	03/29/1996	LEE	Manitoba	5
	03/29/1996	GOUNOT	Ontario-South	3
	03/29/1996	GOUNOT	Quebec	1
	03/29/1996	GOUNOT	Manitoba	7
	03/30/1996	LUCCHESSI	Ontario-South	1
	03/30/1996	LUCCHESSI	Quebec	2
	03/30/1996	LUCCHESSI	Manitoba	1
	03/30/1996	LEE	Ontario-South	7
	03/30/1996	LEE	Ontario-North	3
	03/30/1996	LEE	Quebec	7

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
	03/30/1996	LEE	Manitoba	4
	03/30/1996	GOUNOT	Ontario-South	2
	03/30/1996	GOUNOT	Quebec	18
	03/30/1996	GOUNOT	Manitoba	1
	03/31/1996	LUCCHESSI	Manitoba	1
	03/31/1996	LEE	Ontario-South	14
	03/31/1996	LEE	Ontario-North	3
	03/31/1996	LEE	Quebec	7
	03/31/1996	LEE	Manitoba	3
	03/31/1996	GOUNOT	Ontario-South	2
	03/31/1996	GOUNOT	Quebec	1
	04/01/1996	LUCCHESSI	Ontario-South	3
	04/01/1996	LUCCHESSI	Manitoba	1
	04/01/1996	LEE	Ontario-South	8
	04/01/1996	LEE	Ontario-North	-
	04/01/1996	LEE	Quebec	8
	04/01/1996	LEE	Manitoba	9
	04/01/1996	GOUNOT	Ontario-South	3
	04/01/1996	GOUNOT	Ontario-North	1
	04/01/1996	GOUNOT	Quebec	3
	04/01/1996	GOUNOT	Manitoba	7

STAFF Table

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	smallint not null	varchar(9)	smallint	char(5)	smallint	dec(7,2)	dec(7,2)
Desc:	Employee number	Employee name	Department number	Job type	Years of service	Current salary	Commission
Values:	10	Sanders	20	Mgr	7	18357.50	-
	20	Pernal	20	Sales	8	18171.25	612.45
	30	Marenghi	38	Mgr	5	17506.75	-
	40	O'Brien	38	Sales	6	18006.00	846.55
	50	Hanes	15	Mgr	10	20659.80	-
	60	Quigley	38	Sales	-	16808.30	650.25
	70	Rothman	15	Sales	7	16502.83	1152.00
	80	James	20	Clerk	-	13504.60	128.20
	90	Koonitz	42	Sales	6	18001.75	1386.70
	100	Plotz	42	Mgr	7	18352.80	-
	110	Ngan	15	Clerk	5	12508.20	206.60
	120	Naughton	38	Clerk	-	12954.75	180.00

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
	130	Yamaguchi	42	Clerk	6	10505.90	75.60
	140	Fraye	51	Mgr	6	21150.00	-
	150	Williams	51	Sales	6	19456.50	637.65
	160	Molinare	10	Mgr	7	22959.20	-
	170	Kermisch	15	Clerk	4	12258.50	110.10
	180	Abrahams	38	Clerk	3	12009.75	236.50
	190	Sneider	20	Clerk	8	14252.75	126.50
	200	Scoutten	42	Clerk	-	11508.60	84.20
	210	Lu	10	Mgr	10	20010.00	-
	220	Smith	51	Sales	7	17654.50	992.80
	230	Lundquist	51	Clerk	3	13369.80	189.65
	240	Daniels	10	Mgr	5	19260.25	-
	250	Wheeler	51	Clerk	6	14460.00	513.30
	260	Jones	10	Mgr	12	21234.00	-
	270	Lea	66	Mgr	9	18555.50	-
	280	Wilson	66	Sales	9	18674.50	811.50
	290	Quill	84	Mgr	10	19818.00	-
	300	Davis	84	Sales	5	15454.50	806.10
	310	Graham	66	Sales	13	21000.00	200.30
	320	Gonzales	66	Sales	4	16858.20	844.00
	330	Burke	66	Clerk	1	10988.00	55.50
	340	Edwards	84	Sales	7	17844.00	1285.00
	350	Gafney	84	Clerk	5	13030.50	188.00

STAFFG Table

Note: STAFFG is only created for double-byte code pages.

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	smallint not null	vargraphic(9)	smallint	graphic(5)	smallint	dec(9,0)	dec(9,0)
Desc:	Employee number	Employee name	Department number	Job type	Years of service	Current salary	Commission
Values:	10	Sanders	20	Mgr	7	18357.50	-
	20	Pernal	20	Sales	8	18171.25	612.45
	30	Marenghi	38	Mgr	5	17506.75	-
	40	O’Brien	38	Sales	6	18006.00	846.55
	50	Hanes	15	Mgr	10	20659.80	-
	60	Quigley	38	Sales	-	16808.30	650.25

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
	70	Rothman	15	Sales	7	16502.83	1152.00
	80	James	20	Clerk	-	13504.60	128.20
	90	Koonitz	42	Sales	6	18001.75	1386.70
	100	Plotz	42	Mgr	7	18352.80	-
	110	Ngan	15	Clerk	5	12508.20	206.60
	120	Naughton	38	Clerk	-	12954.75	180.00
	130	Yamaguchi	42	Clerk	6	10505.90	75.60
	140	Fraye	51	Mgr	6	21150.00	-
	150	Williams	51	Sales	6	19456.50	637.65
	160	Molinare	10	Mgr	7	22959.20	-
	170	Kermisch	15	Clerk	4	12258.50	110.10
	180	Abrahams	38	Clerk	3	12009.75	236.50
	190	Sneider	20	Clerk	8	14252.75	126.50
	200	Scoutten	42	Clerk	-	11508.60	84.20
	210	Lu	10	Mgr	10	20010.00	-
	220	Smith	51	Sales	7	17654.50	992.80
	230	Lundquist	51	Clerk	3	13369.80	189.65
	240	Daniels	10	Mgr	5	19260.25	-
	250	Wheeler	51	Clerk	6	14460.00	513.30
	260	Jones	10	Mgr	12	21234.00	-
	270	Lea	66	Mgr	9	18555.50	-
	280	Wilson	66	Sales	9	18674.50	811.50
	290	Quill	84	Mgr	10	19818.00	-
	300	Davis	84	Sales	5	15454.50	806.10
	310	Graham	66	Sales	13	21000.00	200.30
	320	Gonzales	66	Sales	4	16858.20	844.00
	330	Burke	66	Clerk	1	10988.00	55.50
	340	Edwards	84	Sales	7	17844.00	1285.00
	350	Gafney	84	Clerk	5	13030.50	188.00

Sample Files with BLOB and CLOB Data Type

This section shows the data found in the EMP_PHOTO files (pictures of employees) and EMP_RESUME files (resumes of employees).

Quintana Photo

Figure 7. Delores M. Quintana

Quintana Resume

The following text is found in the db200130.asc and db200130.scr files.

Resume: Delores M. Quintana

Personal Information

Address:	1150 Eglinton Ave Mellonville, Idaho 83725
Phone:	(208) 555-9933
Birthdate:	September 15, 1925
Sex:	Female
Marital Status:	Married
Height:	5’2″
Weight:	120 lbs.

Department Information

Employee Number:	000130
Dept Number:	C01
Manager:	Sally Kwan
Position:	Analyst
Phone:	(208) 555-4578
Hire Date:	1971-07-28

Education

1965	Math and English, B.A. Adelphi University
1960	Dental Technician Florida Institute of Technology

Work History

10/91 - present	Advisory Systems Analyst Producing documentation tools for engineering department.
12/85 - 9/91	Technical Writer Writer, text programmer, and planner.
1/79 - 11/85	COBOL Payroll Programmer Writing payroll programs for a diesel fuel company.

Interests
- Cooking
- Reading
- Sewing
- Remodeling

Nicholls Photo

Figure 8. Heather A. Nicholls

Nicholls Resume

The following text is found in the db200140.asc and db200140.scr files.

Resume: Heather A. Nicholls

Personal Information

Address:	844 Don Mills Ave Mellonville, Idaho 83734
Phone:	(208) 555-2310
Birthdate:	January 19, 1946
Sex:	Female
Marital Status:	Single
Height:	5’8″
Weight:	130 lbs.

Department Information

Employee Number:	000140
Dept Number:	C01
Manager:	Sally Kwan
Position:	Analyst
Phone:	(208) 555-1793
Hire Date:	1976-12-15

Education

1972	Computer Engineering, Ph.D. University of Washington
1969	Music and Physics, M.A. Vassar College

Work History

2/83 - present	Architect, OCR Development Designing the architecture of OCR products.
12/76 - 1/83	Text Programmer Optical character recognition (OCR) programming in PL/I.
9/72 - 11/76	Punch Card Quality Analyst Checking punch cards met quality specifications.

Interests

- Model railroading
- Interior decorating
- Embroidery
- Knitting

Adamson Photo

Figure 9. Bruce Adamson

Adamson Resume

The following text is found in the db200150.asc and db200150.scr files.

Resume: Bruce Adamson

Personal Information

Address:	3600 Steeles Ave Mellonville, Idaho 83757
Phone:	(208) 555-4489
Birthdate:	May 17, 1947
Sex:	Male
Marital Status:	Married
Height:	6’0″
Weight:	175 lbs.

Department Information

Employee Number:	000150
Dept Number:	D11
Manager:	Irving Stern
Position:	Designer
Phone:	(208) 555-4510

Hire Date:	1972-02-12

Education

1971	Environmental Engineering, M.Sc. Johns Hopkins University
1968	American History, B.A. Northwestern University

Work History

8/79 - present	Neural Network Design Developing neural networks for machine intelligence products.
2/72 - 7/79	Robot Vision Development Developing rule-based systems to emulate sight.
9/71 - 1/72	Numerical Integration Specialist Helping bank systems communicate with each other.

Interests
- Racing motorcycles
- Building loudspeakers
- Assembling personal computers
- Sketching

Walker Photo

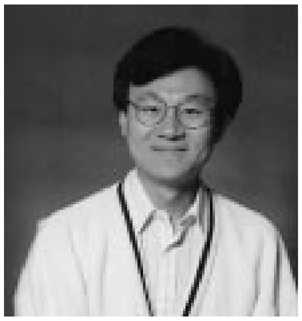


Figure 10. James H. Walker

Walker Resume

The following text is found in the db200190.asc and db200190.scr files.

Resume: James H. Walker

Personal Information	
Address:	3500 Steeles Ave Mellonville, Idaho 83757
Phone:	(208) 555-7325
Birthdate:	June 25, 1952
Sex:	Male
Marital Status:	Single
Height:	5’11″
Weight:	166 lbs.
Department Information	
Employee Number:	000190
Dept Number:	D11
Manager:	Irving Stern
Position:	Designer
Phone:	(208) 555-2986
Hire Date:	1974-07-26
Education	
1974	Computer Studies, B.Sc. University of Massachusetts
1972	Linguistic Anthropology, B.A. University of Toronto
Work History	
6/87 - present	Microcode Design Optimizing algorithms for mathematical functions.
4/77 - 5/87	Printer Technical Support Installing and supporting laser printers.
9/74 - 3/77	Maintenance Programming Patching assembly language compiler for mainframes.

Interests
- Wine tasting
- Skiing

- Swimming
- Dancing

Appendix B. Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the

IBM Director of Licensing
IBM CorporationNorth Castle Drive
Armonk, NY 10504-1785
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Limited
Department 071
1150 Eglinton Ave. East
North York, Ontario
M3C 1H7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries:

ACF/VTAM
ADSTAR
AISPO
AIX
AIXwindows
AnyNet
APPN
AS/400
CICS
C Set++
C/370
DATABASE 2
DatagLANce
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
Distributed Relational Database Architecture
DRDA
Extended Services
FFST
First Failure Support Technology
IBM
IMS
Lan Distance
MVS/ESA
MVS/XA
NetView
OS/400
OS/390
OS/2
PowerPC
QMF
RACF
RISC System/6000
SAA
SP
SQL/DS
SQL/400
S/370
System/370
System/390
SystemView
VisualAge
VM/ESA
VSE/ESA
VTAM
WIN-OS/2

Trademarks of Other Companies

The following terms are trademarks or registered trademarks of the companies listed:

C-bus is a trademark of Corollary, Inc.

HP-UX is a trademark of Hewlett-Packard.

Java, HotJava, Solaris, Solstice, and Sun are trademarks of Sun Microsystems, Inc.

Linux is a trademark of Linus Torvalds.

Index

Contacting IBM

This section lists ways you can get more information from IBM.

If you have a technical problem, please take the time to review and carry out the actions suggested by the *Troubleshooting Guide* before contacting DB2 Customer Support. Depending on the nature of your problem or concern, this guide will suggest information you can gather to help us to serve you better.

For information or to order any of the DB2 Universal Database products contact an IBM representative at a local branch office or contact any authorized IBM software remarketer.

Telephone

If you live in the U.S.A., call one of the following numbers:
- 1-800-237-5511 to learn about available service options.
- 1-800-IBM-CALL (1-800-426-2255) or 1-800-3IBM-OS2 (1-800-342-6672) to order products or get general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, see Appendix A of the IBM Software Support Handbook. You can access this document by selecting the "Roadmap to IBM Support" item at: http://www.ibm.com/support/.

Note that in some countries, IBM-authorized dealers should contact their dealer support structure instead of the IBM Support Center.

World Wide Web

http://www.software.ibm.com/data/
http://www.software.ibm.com/data/db2/library/

The DB2 World Wide Web pages provide current DB2 information about news, product descriptions, education schedules, and more. The DB2 Product and Service Technical Library provides access to frequently asked questions, fixes, books, and up-to-date DB2 technical information. (Note that this information may be in English only.)

Anonymous FTP Sites

ftp.software.ibm.com

Log on as anonymous. In the directory /ps/products/db2, you can find demos, fixes, information, and tools concerning DB2 and many related products.

Internet Newsgroups

comp.databases.ibm-db2, bit.listserv.db2-l

These newsgroups are available for users to discuss their experiences with DB2 products.

CompuServe

GO IBMDB2 to access the IBM DB2 Family forums

All DB2 products are supported through these forums.

To find out about the IBM Professional Certification Program for DB2 Universal Database, go to http://www.software.ibm.com/data/db2/db2tech/db2cert.html

Part Number: CT6N3NA

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

CT6N3NA

SC09-2856-00