Database Systems for Management Third edition

James F. Courtney

David B. Paradice

Kristen L. Brewer

Julia C. Graham

For any questions about this text, please email: drexel@uga.edu

The Global Text Project is funded by the Jacobs Foundation, Zurich, Switzerland

CHAPTER 7
THE STRUCTURED QUERY LANGUAGE

The previous chapter laid the foundation for understanding the relational database model. This chapter builds on that foundation by describing the Structured Query Language (SQL) for relational databases. SQL has been adopted by the American National Standards Institute (ANSI) as the standard relational database language. Numerous implementations of SQL exist for machines ranging from personal computers to mainframes. It is perhaps the most widely used database language. SQL may be used either interactively or it may be embedded in a host programming language. Both types of access will be discussed, beginning with interactive use.

The Structured Query Language.

SQL is a relational transformation language. It provides ways to construct relations and manipulate the data in them. The result of a transformation operation is always another relation (which may have only one row and column).

SQL was developed in IBM research labs during the 1970's. It was the outgrowth of a previous language called SEQUEL (for Structured English Query Language). Many people still pronounce the acronym SQL as "sequel." IBM did not release a commercial product supporting the language until 1981, when SQL/DS was made available for the DOS/VSE environment, and later for the VM/CMS environment. At that time, Oracle Corporation had been marketing a relational system with an SQL interface for several months. Another widely used IBM package, DB2, was released for the MVS environment in 1983.

The American National Standards Institute developed specifications for the SQL language and in 1986 adopted SQL as its official relational language. Standard SQL had an immediate impact on the database marketplace. As early as 1987, over 50 implementations of SQL were available commercially. In addition to those mentioned above some of the more popular packages include XDB from Software Systems Technology, INGRES developed by Relational Technology, XQL now marketed by Novelle and Ashton Tate's dBASE IV.

Standard SQL is important because it allows training of both programmers and end-users in a language that is common to many computer systems and organizational environments. Also, database applications can be ported much more easily to different environments, and database in different computer systems can be accessed more readily. Standards also facilitate the development of portable application packages (such as accounting systems) that operate in different database environments.

Unfortunately, as any other standard language, SQL is not without its problems. As you will see in the examples, many queries are readily expressible in SQL, but some are very difficult to express. We will illustrate SQL queries using the database shown in Figure 7-1. First we will show how the relational algebra operations of selection, projection, and join are performed. More extensive features of the query language are presented after that, including how SQL is embedded in host language programs.

Projection

Projection in SQL is accomplished using the command keywords SELECT and FROM. The use of the word SELECT for projection may seem confusing, since it seems to suggest the selection operation of relational algebra. In fact, selection is performed using a variation of this command.

To perform a projection, one follows the SELECT keyword with the desired attributes. The order in which the attributes are listed determines the order in which they appear in the result. The FROM keyword is then followed by the name of the relation from which the attributes should be taken.

Following the examples in the previous chapter, the projection of EMPLOYEE over NAME and E_NUMBER is specified as

```
SELECT NAME, E_NUMBER
FROM EMPLOYEE
```

The result of this command is shown in Figure 7-2a. Not all implementations of SQL include the relation name in the result as in Figure 7-2a. We include the name, however, to help clarify how the result is formed.

The projection of EMPLOYEE over DEPARTMENT is

```
SELECT DEPARTMENT
FROM EMPLOYEE
```

The result of this command is shown in Figure 7-2b. Note that the result contains duplicate tuples. SQL does not actually impose the closure property. In order to remove duplicate tuples from the result of a SELECT command, one must include the DISTINCT keyword. The command

```
SELECT DISTINCT DEPARTMENT
FROM EMPLOYEE
```

results in only one occurrence of each department name, giving the result in Figure 7-2c.

Projection may also be done on the basis of values computed from numeric database items, using the operators +, -, / and *. As in other languages, multiplication and division are performed before addition and subtraction. This order may be controlled by using parentheses. To illustrate, suppose we want a list of the difference between actual and expected cost for each project. The appropriate query is

```
SELECT NAME, ACTUAL_COST - EXPECTED_COST
FROM PROJECT
```

The result is shown in Figure 7-3a.

If we wanted to compute percent of actual to expected, the query would be:

```
SELECT NAME, ACTUAL_COST / EXPECTED_COST * 100.0
FROM PROJECT
```

See Figure 7-3b for the result.

Selection

Selection is performed with combinations of the keywords SELECT, FROM, and WHERE. The simplest selection is a retrieval of all attributes from a relation. Retrieving an entire relation, such as the EMPLOYEE relation, can be accomplished by using the command

```
SELECT *
FROM EMPLOYEE
```

The asterisk (*) indicates that all attributes in the relation should be retrieved.

The WHERE keyword is used to specify conditionS for selecting tuples from a relation. To retrieve the tuples for all employees in the finance department, one would use the following command, producing the result in Figure 7-4:

```
SELECT *
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
```

Combining Projection and Selection

Combining projection and selection is straightforward in SQL. To retrieve only the names of the employees in the finance department (as shown in Figure 7-5a) one would specify:

```
SELECT NAME
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
```

To select finance employee names and numbers, the query is (Figure 7-5b gives the result):

```
SELECT NAME, E_NUMBER
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
```

Finally, suppose we want to list employee number before name in the previous query. All we do is interchange the position of NAME and E_NUMBER, as follows (see Figure 7-5c):

```
SELECT E_NUMBER, NAME
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
```

Other examples of combining projection and selection will be given throughout the chapter.

Join

A join can be performed in SQL by using more detailed SELECT, FROM, and WHERE clauses. To illustrate, suppose we want a list of employee names and titles. To associate employee names with employee titles, we must join the EMPLOYEE and TITLE relations. The SQL command for this is

```
SELECT *
FROM EMPLOYEE, TITLE
WHERE EMPLOYEE.E_NUMBER = TITLE.E_NUMBER
```

The FROM keyword is followed by the names of the relations to be joined. The WHERE keyword is followed by the attributes over which the join is to occur. The attribute names are prefixed ("qualified") with the name of the relation that contains the attribute so that all attribute names are uniquely identified. The relation and attribute names are separated by a period. The result of this command is shown in Figure 7-6.

To illustrate a slightly more complex join, suppose we want to know the names of all the senior consultants. This requires data from both the TITLE and EMPLOYEE relations, joined over employee number. This is done in SQL as follows:

```
SELECT NAME
FROM EMPLOYEE, TITLE
```

```
WHERE EMPLOYEE.E_NUMBER = TITLE.E_NUMBER
AND CURRENT_TITLE = 'Senior Consultant'
```

The results of the query above are shown in Figure 7-7a. Note the use of the AND keyword to link the two conditions in the WHERE clause. AND is discussed further below. There is no need to qualify NAME in the SELECT clause because NAME is not an attribute of TITLE, so there is no ambiguity. Of course qualifying NAME as EMPLOYEE.NAME would cause no problems.

Finally, for an example that draws on three tables (a "3-way" join), suppose we want the names and skills of all Senior consultants. To do this, the previous query may be extended as shown below (results are shown in Figure 7-7b):

```
SELECT NAME, SKILL
FROM EMPLOYEE, TITLE, EXPERTISE
WHERE EMPLOYEE.E_NUMBER = TITLE.E_NUMBER
AND CURRENT_TITLE = 'Senior Consultant'
AND EXPERTISE.E_NUMBER = TITLE.E_NUMBER
```

The IN Keyword and Joins via Subqueries

WHERE clauses can contain other more powerful keywords. One example is the keyword IN. IN is used to specify a list of values an attribute can assume. To retrieve the employee numbers for the employees whose skill is either taxation or management, one could specify

```
SELECT E_NUMBER
FROM EXPERTISE
WHERE SKILL IN ('Taxation', 'Management')
```

The result is shown in Figure 7-8.

The IN keyword can also be followed by another SELECT-FROM-WHERE keyword combination (called a subquery or nested query). To obtain the skills of employees that have had the prior job title "Research analyst" requires data from two relations. The following command demonstrates how to achieve this result, which is shown in Figure 7-9a.

```
SELECT SKILL
FROM EXPERTISE
WHERE E_NUMBER IN
    (SELECT E_NUMBER
    FROM PRIOR_JOB
    WHERE PRIOR_TITLE = 'Research analyst')
```

Think of the execution of this statement in terms of two loops in a program, one embedded in the other. In the inner loop, employee numbers are selected for employees whose prior job title is "Research analyst." In the outer loop, the employee numbers provided by the inner loop are used to select the skills shown in the result. Some people find it easier to think of the clause after the IN keywords as a subroutine that returns a set of values (in this case, the employee numbers).

The NOT keyword can be used to negate the IN clause by specifying NOT IN. The skills of the employees who have not had a prior "Research analyst" title are found by using

```
SELECT SKILL
FROM EXPERTISE
```

```
WHERE E_NUMBER NOT IN
      (SELECT E_NUMBER
      FROM PRIOR_JOB
      WHERE PRIOR_TITLE = 'Research analyst')
```

This gives the result shown in Figure 7-9b.

Joins may also be done by using subqueries. For example, the results in Figure 7-7a may be produced by using:

```
SELECT NAME
FROM EMPLOYEE
WHERE E_NUMBER IN
      (SELECT E_NUMBER
      FROM TITLE
      WHERE CURRENT_TITLE = 'Senior Consultant')
```

To produce the results in Figure 7-7b with a subquery, use

```
SELECT NAME, SKILL
FROM EMPLOYEE, EXPERTISE
WHERE EMPLOYEE.E_NUMBER = EXPERTISE.E_NUMBER
AND EXPERTISE, E_NUMBER IN
      (SELECT E_NUMBER
      FROM TITLE
      WHERE CURRENT_TITLE = 'Senior Consultant')
```

The choice of using AND's or subqueries to perform joins is really a matter of personal preference. A third way of performing joins with the keyword EXITS is discussed later in the chapter.

WHERE clauses can contain more complex specifications using the connectors AND and OR, and the comparison operators =, >, <, <> (not equal), <=, and >=. For example, suppose one needs to know which projects managed by Kanter are over budget. To achieve the results shown in Figure 7-10, one would specify the following SQL command:

```
SELECT NAME
FROM PROJECT
WHERE Manager = 'Kanter' AND Actual Cost > Expected Cost
```

To gain an understanding of the power SQL provides the user, consider what would be necessary to program this example. Figure 7-11 shows a pseudocode program to perform this retrieval. Obviously the SQL version is much simpler to construct, especially for the novice user.

The WHERE clause can be made as specific as needed by using the connective keywords AND and OR. The following command retrieves the names of all projects over budget that are managed by either Kanter or Baker:

```
SELECT NAME
FROM PROJECT
WHERE ACTUAL_COST > EXPECTED_COST
      AND ((MANAGER = 'Kanter') OR (MANAGER = 'Baker'))
```

Figure 7-12a shows the result of this command.

Standard SQL also supports the keywords BETWEEN and LIKE. BETWEEN is used to specify a range of values for a condition. Suppose we want a list of all projects where the expected cost is > = 4000 and < = 8000. Using BETWEEN, we have:

```
SELECT NAME
FROM PROJECT
Where EXPECTED_COST BETWEEN 4000 AND 8000.
```

Results are shown in Figure 7-12b. Notice that the lower value (4000) precedes the upper value and that values equal to the lower or upper limit are included in the result.

LIKE is used to indicate a pattern to match, instead of an exact character string, when searching textual items. The % symbol is used to indicate a string of any characters of arbitrary length (including zero length), and the underscore symbol (_) is used to indicate one arbitrary character. So if we want a list of all employees whose name begins with A, the query is:

```
SELECT NAME
FROM EMPLOYEE
WHERE NAME LIKE 'A%'
```

The result would be Adams. If we want employees names that begin with A and are 3 characters long, we would substitute 'A__' for 'A%' above. If we want employees whose name ends in r, we would use '%r'.

WHERE clauses may also be based on computed values. Suppose we want a list of projects that are 20 percent or more over budget (expected cost). The query is:

```
SELECT NAME
FROM PROJECT
WHERE (ACTUAL_COST - EXPECTED_COST)/
    EXPECTED_COST * 100.0 > 20.0
```

Results are shown in Figure 7-12c.

Sorting Results

SQL provides an ORDER BY clause to sort results on one or more columns called sort keys. Output is displayed in ascending (ASC) order, unless descending order is specified by using DESC.

For example, to display the results in Figure 7-13 in alphabetical (ascending) order, one would use:

```
SELECT NAME
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
ORDER BY NAME
```

If we wanted these names in inverse alphabetical order, the last line above would be replaced with:

```
ORDER BY NAME DESC
```

To illustrate the use of two sort keys, suppose we want to rearrange the EMPLOYEE table so that department names are alphabetized, employee names are listed alphabetically within departments, and the column order is DEPARTMENT, NAME, E_NUMBER. The appropriate query is (results are shown in Figure 7-14):

```
SELECT DEPARTMENT, NAME, E_NUMBER
FROM EMPLOYEE
ORDER BY DEPARTMENT, NAME
```

Finally, rather than using column names to specify sort keys, the relative position of the key in the SELECT list may be used. Thus, the following query produces the same result as the query above.

```
SELECT DEPARTMENT, NAME, E_NUMBER
FROM EMPLOYEE
ORDER BY 1,2
```

Aggregate Functions

SQL includes a set of built-in functions for computing aggregate values of numeric columns. Aggregate functions (sometimes called column functions) include SUM, AVG, MAX, MIN and COUNT. With the exception of the special case COUNT (*), these functions operate on one, numerically-valued column of a table and return a single number (a "scalar") as the result. COUNT (*) is a special case because it operates on all columns of a table.

The COUNT function provides the number of tuples in a relation that satisfy the conditions (if any) specified in a WHERE clause. In the simplest case, there are no conditions. The command:

```
SELECT COUNT (*)
FROM PROJECTS
```

gives the result 7, the number of projects currently in progress at the company.

Because SQL does not automatically suppress duplicate tuples, the command

```
SELECT COUNT (DEPARTMENT)
FROM EMPLOYEES
```

gives the result 7. Determining the number of unique department names requires use of the DISTINCT keyword. The proper command for this information is

```
SELECT COUNT (DISTINCT DEPARTMENT)
FROM EMPLOYEES
```

This command produces the result 2.

The remaining aggregate functions operate on numeric values only. To see how these functions operate, consider the PROJECT relation, reproduced in Figure 7-15a. The SQL SUM function can be used to provide a total of the cost values. To calculate the total expected cost of the projects in progress, for example, the following command is used:

```
SELECT SUM (EXPECTED_COST)
```

```
FROM PROJECT
```

This produces the result $33,500. The average expected cost can be computed by using the AVG function:

```
SELECT AVG (EXPECTED_COST)
FROM PROJECT
```

This command gives the result $4,785.71 (to the nearest cent). In order to find the projects with above-average expected cost, one could use

```
SELECT *
FROM PROJECT
WHERE EXPECTED_COST > (SELECT AVG (EXPECTED_COST)
          FROM PROJECT)
```

This produces the result shown in Figure 7-15b. Maximum and minimum values can be calculated using MAX and MIN functions, respectively. The project with the maximum actual cost accrued can be obtained using

```
SELECT *
FROM PROJECT
WHERE ACTUAL_COST = (SELECT MAX (ACTUAL_COST)
          FROM PROJECT)
```

The result is shown in Figure 7-15c. Similarly, the project with the least EXPECTED_COST can be determined by using

```
SELECT *
FROM PROJECT
WHERE EXPECTED_COST = (SELECT MIN (EXPECTED_COST)
          FROM PROJECT)
```

The result of this query is shown in Figure 7-15d. Finally, these functions may be combined. For example, the number of projects with expected costs that exceed the average expected cost can be found as follows:

```
SELECT COUNT (*)
FROM PROJECT
WHERE EXPECTED_COST > (SELECT AVG (EXPECTED_COST)
          FROM PROJECT)
```

The result indicates that there are three projects with expected cost exceeding the average expected cost.

Grouping Results

Suppose we want to know the total amount spent (actual cost) on projects managed by each person in the PROJECT relation. We could do this for Baker, for example, by using:

```
SELECT SUM (ACTUAL_COST)
FROM PROJECT
WHERE MANAGER = 'Baker'
```

This is very cumbersome because we would have to perform a SELECT for each manager.

SQL provides a GROUP BY clause to aggregate data on the basis of categories of data such as different manager's names. To group the output by manager, the following command would be used (the manager's name is also retrieved):

```
SELECT MANAGER, SUM (ACTUAL_COST)
FROM PROJECT
GROUP BY MANAGER
```

The results of this query are shown in Figure 7-16a. If we want the result sorted, we would use the ORDER BY clause, say as follows:

```
SELECT MANAGER, SUM (ACTUAL_COST)
FROM PROJECT
GROUP BY MANAGER
ORDER BY MANAGER
```

Notice that GROUP BY precedes ORDER BY, which is a requirement of SQL. Results are shown in Figure 7-16b.

There may be situations where we would like to specify conditions on groups (rather than rows as the WHERE clause does). We might want the information on the managers above, only if total expenditures exceed $5000, for example. The HAVING clause is used with the GROUP BY to specify group conditions similar to the way WHERE is used to specify tuple conditions. The appropriate query for managers who have spent more than $5000 is:

```
SELECT MANAGER, SUM (ACTUAL_COST)
FROM PROJECT
GROUP BY MANAGER
HAVING SUM (ACTUAL_COST) > 5000
ORDER BY MANGER
```

Results are illustrated in Figure 7-16c.

WHERE can be used before GROUP BY and HAVING to restrict rows that qualify. Suppose we want the total amount by which actual cost exceeds expected cost by manager, but only on those projects that are over budget. The query is:

```
SELECT MANGER, SUM (ACTUAL_COST - EXPECTED_COST)
FROM PROJECT
WHERE ACTUAL_COST > EXPECTED_COST
GROUP BY MANAGER
```

Figure 7-16d shows the results of this query.

Other Functions

Another set of functions, sometimes called table functions, operate on more than one column. DISTINCT is actually a table function which takes a table as input and returns a table with any duplicate rows removed.

UNION. UNION is a table function which performs a relational algebra union operation. The syntax is X UNION Y where X and Y are union-compatible tables. DISTINCT and UNION both return entire tables.

EXISTS. EXISTS is a logical operator that returns the value "true" or "false." The EXISTS keyword usually follows a SELECT-FROM-WHERE combination. This function returns the value "true" whenever a row satisfying the WHERE clause exists; otherwise it returns the value "false."

As an example, consider a situation in which one wishes to determine which employees have more than one skill. The following SQL query can be used to achieve the relation shown in Figure 7-17.

```
SELECT DISTINCT E_NUMBER
FROM EXPERTISE FIRST
WHERE EXISTS
        SELECT *
        FROM EXPERTISE SECOND
        WHERE FIRST.E_NUMBER = SECOND.E_NUMBER
        AND FIRST.SKILL NOT= SECOND.SKILL
```

Here the FROM keyword is followed by a relation name and another word. The word following the relation name can be used as an alias or synonym for the relation name. This is done to allow the relation to be referenced in two different ways.

You may find that imagining two identical occurrences of the EXPERTISE relation is helpful in understanding this command. Refer to the first occurrence as FIRST and to the second as SECOND. The query above takes each tuple from FIRST and compares it to every tuple in SECOND. When there is a match on the employee number and the skills are not the same, then the tuple from SECOND satisfies the condition in the second (lower) WHERE clause. This sets the value of EXISTS to "true." Because the value of EXISTS is "true," the employee number from the FIRST relation is included in the result.

Thus far we have assumed that the database in Figure 7-1 has already been defined and that the data has been entered into it. The next section describes the database definition features of SQL, how data is entered and maintained, and security features including views and access privileges.

Defining Databases in SQL

Defining a database in SQL is a rather straightforward process. The command CREATE TABLE is used to name each relation in the database, to name each attribute in the relation, to define the data type for each attribute, and to express certain integrity constraints. For example, the EMPLOYEE relation in Figure 7-1 would be defined as follows:

```
CREATE TABLE EMPLOYEE
    (NAME CHAR (12)),
    E_NUMBER SMALLINT NOT NULL UNIQUE,
    DEPARTMENT CHAR (12))
```

In the command above, the indentation is to improve readability and is not required. CHAR means character data and the number in parentheses following CHAR is the maximum number of characters. SMALLINT means an integer in the range - 32768 to +32767. The key words NOT NULL and UNIQUE associated with E_NUMBER are used because this is the key attribute for the relation and means that E_NUMBER must have a unique value for any tuple inserted into the database.

For another example, consider the EXPERTISE relation, which would be defined as follows:

```
CREATE TABLE EXPERTISE
    (E_NUMBER SMALLINT NOT NULL,
    SKILL CHAR (15) NOT NULL,
    UNIQUE (E_NUMBER, SKILL))
```

Here E_NUMBER and SKILL appear at the end of the command and form a concatenated key, so both are NOT NULL, and their combination must be unique.

Finally, consider the PROJECT relation, which would be specified with the statement:

```
CREATE TABLE PROJECT
     (NAME CHAR (25),
     P_NUMBER SMALLINT NOT NULL, UNIQUE,
     MANAGER CHAR (12)
     ACTUAL_COST INTEGER,
     EXPECTED_COST INTEGER)
```

Above, ACTUAL_COST and EXPECTED_COST have been defined as INTEGER, permitting values in the range -2, 147, 483, 648 to 2, 147, 483, 647. Suppose we wanted to express these attributes in dollars and cents, then the DECIMAL data type could be substituted as follows:

```
ACTUAL_COST DECIMAL (12,2),
EXPECTED_COST DECIMAL (12,2)
```

This statement allows a total of 12 digits, with 2 to the right of the decimal point. One other SQL data type is FLOAT, to specify floating point numbers given in exponential notation.

The CREATE TABLE command results in establishing an empty base table. IN SQL, base tables have a physical representation in storage, although they may not be stored exactly as files with record occurrences as described in the table format. Use of the terms file and record are thus discouraged as possibly misleading. However, users can always think of base tables as physically existing without worrying about representation details.

Inserting and Deleting Data

The command INSERT INTO is used to add data to base tables. For example, the first row of the EMPLOYEE relation could be added with the statement:

```
INSERT INTO EMPLOYEE
VALUES  (Kanter, 1111, FINANCE)
```

SQL also provides basic capabilities for modifying relations. To insert a new tuple into a relation, one specifies the INSERT INTO keywords followed by the name of the relation; this is followed by the data belonging in the new tuple, For example,

```
INSERT INTO EMPLOYEE
VALUES (Farmer, 1006, Accounting)
```

adds a tuple for a new employee to the EMPLOYEE relation. If the department in which the new employee will be working is unknown, an alternate form of the update can be used:

```
INSERT INTO EMPLOYEE (NAME, E_NUMBER)
VALUES (Farmer, 1006)
```

This results in a null value for the department attribute.

It is also possible to insert tuples from another relation. This is accomplished by specifying a SELECT-

FROM-WHERE block after the INSERT INTO keywords. To combine all prior job titles with the current titles, one can use

```
INSERT INTO TITLE
     SELECT *
     FROM PRIOR_JOB
```

A WHERE clause can also be specified to select particular employee numbers. Of course, the relations must be union compatible for this operation to be valid.
Deletion

The DELETE and WHERE keywords are used to remove tuples from a relation. The DELETE keyword is followed by the name of the relation containing the tuple to be deleted. The WHERE keyword is used in its normal manner--to specify a condition that selects a tuple or set of tuples from a relation.

To delete Adams from the EMPLOYEE relation, one can use this command:

```
DELETE EMPLOYEE
WHERE Name = 'Adams'
```

SQL is not a particularly difficult language to master; however, in some cases queries can be rather tricky. Since good relational designs often end p separating the data into numerous tables, it is often necessary to join several tables to get the desired result. This can be quite cumbersome. One way of avoiding this problem is through the use of user views, which are discussed next.

Defining Views

User views, which have no physical analog in the stored database, may be defined over one or more base tables, or previously defined views. Views may involve relational algebra projection, selection, or join. They may be used as a form of security device to hide data from certain users, or they may be used to simplify access to the database.

Suppose we have a user who needs only to know project names and numbers. This requires a projection operation on the PROJECT table. A view can be created for this user with the following command:

```
CREATE VIEW PROJECT_LIST AS
     SELECT NAME, P_NUMBER
     FROM PROJECT
```

Users authorized to use this view (see the section on security for a discussion of authorization) may access a virtual relation called PROJECT-LIST with the columns NAMES and P-Number. If we want to name the columns NAME and NUMBER, the appropriate statement is:

```
CREATE VIEW PROJECT-LIST (NAME, NUMBER) AS
     SELECT NAME, P_NUMBER
     FROM PROJECT
```

Any SELECT clause may be used in defining a view, including GROUP BY and HAVING, which are discussed later. If, for example, the user above were only allowed to access projects managed by Baker or Kanter, we would have:

```
CREATE VIEW PROJECT_LIST AS
```

```
SELECT NAME, P_NUMBER
FROM PROJECT
WHERE MANAGER = 'Baker'
OR MANAGER = 'Kanter'
```

Equivalently, WHERE MANAGER IN ('Baker', 'Kanter') could be substituted for WHERE and OR above.

To illustrate a join, suppose a user needs access to employee names and projects to which that employee is assigned. This required a join of the ASSIGNMENT and EMPLOYEE tables and a projection over employee name and project number, as follows:

```
CREATE VIEW NAME_PROJ (NAME, P_NUMBER) AS
    SELECT EMPLOYEE, NAME, ASSIGNMENT, P_NUMBER
    FROM EMPLOYEE, ASSIGNMENT
    WHERE ASSIGNMENT, E_NUMBER = EMPLOYEE, E_NUMBER
```

If this user is really alphabetically inclined and wants both project names and associated projects, the statement would be

```
CREATE VIEW ALPHA_VIEW (EMPLOYEE, PROJECT) AS
    SELECT EMPLOYEE, NAME, PROJECT_NAME
    FROM EMPLOYEE, ASSIGNMENT, PROJECT
    WHERE EMPLOYEE, E_NUMBER = ASSIGNMENT, E_NUMBER
    AND ASSIGNMENT, P_NUMBER = PROJECT, P_NUMBER
```

Note that this is the world-renowned 3-way join.

Security Features

Views can be used as one security device in that certain data can be hidden from selected users. The GRANT key word is used to specify privileges to users on the basis of an **authorization identifier, also called** permission, which identifies users that have access to the database. (We will use the term userid instead of authorization identifier.) Privileges may be granted on base tables or views and include the following: SELECT, INSERT, DELETE, MODIFY, and ALL. The creator of a view or base table holds all privileges on that table.

Let's say we want to allow the person holding user ID YAHOO to retrieve data from the PROJECT table, but not be able to change, add or remove data from the table. The appropriate GRANT statement is:

```
GRANT SELECT ON PROJECT TO YAHOO
```

Other self-explanatory examples are:

```
GRANT SELECT, UPDATE ON EMPLOYEE TO JAGU
GRANT ALL ON ASSIGNMENT TO ADMIN
GRANT INSERT ON SKILL TO PERSONNEL
GRANT SELECT ON ALPH_VIEW TO ALPHAGUY
```

The last GRANT above applies to a previously defined view.

The special userid PUBLIC may be used to grant privileges to any user having access to the database.

So, for example, we might:

GRANT SELECT ON EXPERTISE TO PUBLIC

In a database with student grades we would probably not GRANT UPDATE ON GRADES TO PUBLIC.
All of the examples that have been given thus far have assumed that the user is interacting directly with the database via the SQL processor. The next section shows how SQL statements can be embedded in programs, so that the user interacts with the program, which then interacts with the SQL database to retrieve or update the necessary data.

Embedded SQL.

SQL statements can also be embedded in a host programming language, such as COBOL, FORTRAN, PL/1, or C. The examples will be given in COBOL; however, they have been designed to be simple enough to understand even if you are unfamiliar with that language. The first example program is shown in Figure 7-17. All this program does is accept an employee number from the user, and fetch and display that employee's title as given in the CURRENT_TITLE table. Note that this program deals with only one row of an SQL table at any one time. An example of retrieving multiple rows is given next.

A few preliminary comments are in order before the program is described. The line numbers in the left-most column are just to make it easier to describe the program. They are not part of the program. COBOL programs are divided into divisions, sections, paragraphs, and sentences. There are always four divisions - identification, environment, data and procedure. The identification division names the program and possibly the programmer. The program in Figure 7-17 is named EMPLOYEE-TITLE. The environment division is used to name the host computer system on which the program is to be run. The statement ENVIRONMENT DIVISION. must be present, but everything is optional here and has been omitted in this program.

The data division requires some explanation. It is used to describe any files and variables used in the program. In the case of embedded SQL, "host variables" (COBOL variables in this case), which will hold data retrieved from or sent to the database, must be described first. This is done in a DECLARE SECTION in lines 6-10. Notice that each SQL sentence begins with EXEC SQL and ends with END-EXEC. There are two COBOL host variables in this program, COBOL-E-NUMBER, and COBOL-CURRENT-TITLE. The word COBOL has been prefixed to these variables as a convenience to remind the programmer that these are variables in the host language, and not SQL names. Putting COBOL in front of host variables is not required.

After the host variables have been declared, any tables used in the database must be declared. This is done in lines 11-15. The declaration of tables is very similar to the CREATE TABLE statement in SQL.

Line 16 contains a sentence that must be included in every COBOL program using embedded SQL. It results in the creation of a "communication area" (SQLCA) between the program and the database system. The communication area automatically contains a variable called SQLCODE that is used to indicate the result of SQL operations performed by the program. If, after the execution of an SQL statement, the value of SQLCODE is 0, then the statement executed normally. If the value is negative, then an error occurred and the value of SQLCODE contains the error number.

The actual algorithm of the program begins in the procedure division starting at line 17. The MAIN-PROCEDURE causes paragraphs ACCEPT-ENUMBER and FETCH-TITLE to be run in that order. ACCEPT-ENUMBER (lines 22-24) displays the message "ENTER EMPLOYEE NUMBER:" on the user's screen, and accepts that number into the host variable COBOL-E-NUMBER. The FETCH-TITLE paragraph in lines 25-35 contains an executable SQL statement that retrieves the current title for that employee. Notice that between the EXEC SQL in line 26 and the END-EXEC in line 31, there is an embedded SQL SELECT. The only difference in this and an interactive SELECT is that the host variables to receive the data are named in the INTO clause following SELECT. Also notice that the host variable names are prefixed with a colon (:) to distinguish them from SQL names.

At line 32, SQLCODE is tested. If it is 0, the current title is displayed by line 33, otherwise, an error has occurred and an error message is displayed. In either case, the program terminates at that point.

The example program in Figure 7-18 demonstrates a situation in which more than one row qualifies as a result of an SQL operation. This program accepts a manager's name, determines which projects are over budget for that manager, and displays a list of those projects. Since one manager is responsible for more than one project, several rows may qualify for retrieval. In such an event, the programmer must use a "cursor" to iterate through the processing of each row. A cursor is a pointer or index to the next row to be processed (it has nothing to do with the cursor on the screen). An SQLCODE of 100 is used to indicate that there are no more rows to be processed.

In the program, we have the identification and environment divisions as before. Five host variables are declared in the DECLARE section (lines 7-14). These variables correspond to those in the PROJECT table. The PROJECT table itself is declared in lines 15-22. Line 23 sets up the communication area, and a new type of entry starts at line 24. Here, the cursor is declared (named) and the SQL statement with which it is associated is given. The SQL statement starts at line 26, and when invoked in the procedure division, finds all data in the PROJECT table for the manager whose name is stored in the host variable COBOL-MANAGER. It also sets the cursor to the first row for that manager.

The main procedure starts at line 31, and first accepts the manager's name in the ACCEPT-MANAGER paragraph. Next, the cursor is invoked in the OPEN-CURSOR paragraph by using the statement in line 41. The PROCESS-MAN paragraph actually retrieves and processes the data, one row at a time. At line 43, PROCESS-MAN retrieves the first row of data (the program assumes that at least one row exists for the given manager), then tests SQLCODE to see if it is zero. If so, paragraph TEST-COST is run until all rows have been processed (SQLCODE = 100), and if not, the error routine is performed.

Paragraph TEST-COST compares actual to budgeted cost, and for any projects over budget, displays the project name, manager name and amount of over-run. Then the next row is fetched and processed. When all rows have been processed, control returns to the main routine, the cursor is closed, and the run is terminated.

The final program (Figure 7-19) is an example of an update operation. Here an employee has been promoted and has a new job title, which must be updated in the CURRENT-TITLE relation. In addition, the old title must be added to the list in the PRIOR-JOB table. Since only one row is involved in each of these tables, a cursor is not necessary.

Again, after the identification and environment divisions the host variables are declared, this time in lines 6-11. The TITLE and PRIOR-JOB tables are declared in lines 12-21, and the communication area is established in line 22.

The procedure division starts at line 23. The employee number for the employee who has been promoted is accepted, along with the new title, in the ACCEPT-DATA paragraph in lines 33-37. The employee's old title is found by the GET-OLDTITLE routine in lines 38-44. After the old title is retrieved, SQLCODE is checked in the main procedure at line 27 to make sure everything is ok, then the PRIOR-JOB table is updated in the UPDATE-PRIOR-JOB paragraph. Line 46-48 gives an example of an embedded INSERT operation. Notice that the values to be inserted are given in host variables which are embedded in parentheses following the keyword VALUES.

Control returns to the main procedure, and SQLCODE is tested again (line 29), then UPDATE-CURRENT-TITLE is performed. The update operation is performed in lines 51-55, which are similar to an interactive update, except that the new value and employee number are given in host variables. Finally, SQLCODE is checked again and the program terminates.

While these three examples don't give all the details of using embedded SQL, they do provide enough information to give you an idea of how host programs interface to SQL databases. You should be able to write simple programs such as these after studying the examples carefully.

THE USER'S VIEW

In "The User's View" in Chapter 6, we hinted that implementations of relational database languages are user-oriented. This chapter provides an example of that orientation.

In SQL, we see that only a small number of keywords need be learned to retrieve data from relational databases. Many of the relational algebra operations of Chapter 4 are variations of the SELECT-FROM-WHERE keyword combination. A small number of commands to learn obviously facilitates the learning process. The choice of keywords also helps users to remember them because they are descriptive of the actual operations performed.

In some cases, however, SQL can require rather complex and verbose. The IN key word and nested queries can be quite lengthy. Joins can require complex and lengthy syntax. The last example of SELECT, for example, is not one that most users would be willing to spend the time to master.

Thus in most cases, users cannot be left to themselves after a database has been designed and implemented, even if SQL is available. They still may need assistance in performing some queries.

Metropolitan National Bank

One way of evaluating a database design is to verify that the design supports the decision-making processes that revolve around the database. If you can implement typical queries related to the decision-making process, you have a good chance that your design supports those processes. Listed below are typical queries that must be supported by MetNat's integrated database system. Use your implementation from the last chapter to construct SQL commands that implement each query. If an SQL command alone is inadequate, indicate how the query could be processed using an application program. If you find a query particularly difficult to implement, you may need to revise your implementation from the prior chapter.

Sample Queries
What is the current balance of an account?
Has a recent deposit been credited to an account?
Has a recent check cleared?
When was the last interest amount credited to my saving account?
What is the current interest rate on money market accounts?
How much interest has been credited to an account?
What is the remaining balance on a loan?
How many more payments are outstanding on this loan?
What accounts does a particular customer hold?
What is the maturity date on a particular certificate of deposit?
What is the address of a particular customer?
When was the current balance on an account last calculated?
What penalties have been charged to a particular account?
What are the loan numbers of loans that have been paid back by a particular customer?
What are the loan numbers of outstanding loans held by a particular loan applicant?
What is the credit limit of a particular credit card holder?
How much available credit does a particular credit card customer have?
What is the average monthly balance on a particular credit card account?
How many sick leave days does an employee currently have?
How many vacation days does an employee currently have?

CHAPTER SUMMARY

This chapter has presented features of SQL for retrieving data from relational databases. SQL is a relational transformation language. As such, it provides a user interface consisting of a set of keywords and a basic structure. The basic keyword set consists of SELECT, FROM, and WHERE. The keywords are used in conjunction with relation and attribute names to specify the desired results of relational data manipulation. SQL performs the appropriate relational operations to achieve the desired results, thus insulating the user from the need to learn or even be aware of relational algebra operations such as projection, selection, and join.

Other aspects of SQL for retrieving data include the keywords IN, BETWEEN and LIKE. IN is used to specify a set of values, which may be defined by a nested subquery (another SELECT-FROM-WHERE combination). BETWEEN is used to specify a range of values in a WHERE clause. LIKE is for specifying character patterns, instead of exact character strings to match in a text search.

The ORDER BY clause in SQL is used to sort output. GROUP BY is available for grouping results and HAVING may be used with GROUP BY to specify conditions on group results. The functions COUNT, SUM, MIN, MAX, and AVG provide the ability to compute aggregate values. The EXISTS function returns the value "false" or "true," DISTINCT returns only the unique rows of a table, and UNION performs a relational algebra union operation. Together these features of SQL provide a powerful set of operators for accessing data in relational data bases. relational databases are created User views, defined with the CREATE VIEW operation, may be used to restrict the portion of the database available to a user and may also be used to simplify access by joining tables in a way that is transparent to the user. Additional security features are provided with the GRANT operation, which is used to specify the type of access permitted to a user. Finally, SQL may be embedded in host programming languages to provide for batch updates, formatted screens, and so forth.

QUESTIONS AND EXERCISES

1. What are the characteristics of a relational transformation language?

2. List the SQL keywords discussed in this chapter. Explain how each is used. What usually follows each?

3. Show the SQL commands to perform the following operations on the EMPLOYEE relation in Figure 7-1.
   a. List the employee names.
   b. List the employee numbers.
   c. Retrieve all data on the employee named Clarke.
   d. Retrieve all data on employee number 1005
   e. List employee names beginning with 'C'.
   f. List employee names with the second letter 'd'.
   g. List employee names between 'B' and 'K'.

4. Show the SQL commands to perform the following operations on the PROJECT relation in Figure 7-1.
   a. List the project names.
   b. List the managers of projects.
   c. Provide all data for the projects managed by Yates.
   d. Provide a list of the projects with actual costs less than the expected cost.
   e. Display project names in alphabetical order.
   f. Display project names in inverse alphabetical order.
   g. Display managers and project numbers, with managers in alphabetic order and their corresponding projects in descending numeric order.
   h. Display project name and the ratio of actual to expected cost.
   i. Display project names with actual costs more than twice expected costs.

5. Show the SQL commands to perform the following operations on the EMPLOYEE and ASSIGNMENT relations in Figure 7-1.
   a. List the employee names and numbers for employees in the accounting department.
   b. Find the department in which Dexter works.
   c. List the employee numbers for employees assigned to project 23760.
   d. List the names of finance employees assigned to project 26511.

6. Show the SQL commands to perform the following operations on the ASSIGNMENT, PROJECT, and EMPLOYEE relations in Figure 7-1.
   a. List the actual costs of projects managed by Yates.
   b. Provide all information on the project managed by Kanter.
   c. Provide data on the actual and expected costs of project 28765.
   d. Provide data on the project with the minimum expected cost.
   e. Provide the names of the employees working on the project to secure a new office lease.

7. Show the SQL commands to perform the following operations on the EMPLOYEE and ASSIGNMENT relations in Figure 7-1.
   a. Count the number of employees
   b. Count the number of projects.
   c. Count the number of departments.
   d. Count the number of employees in each department (use GROUP BY).
   e. Limit the results in 7-d to only those departments with more than 3 employees.

8. Repeat Questions 7a-c, counting only unique tuples.

9. Show the SQL commands to perform the following operations on the EXPERTISE relation in Figure 7-1.
   a. Count the number of employees.
   b. Count the number of unique skills.
   c. Count the number of employees with stock market training.

10. Show the SQL commands to perform the following operations on the PROJECT relation in Figure 7-1.
    a. Find the project with the maximum cost.
    b. Find the projects where actual cost is less than expected cost.
    c. Find the projects with actual costs less that the average actual cost.

11. Create the following:

    a. A view which shows employee names and associated skills.
    b. A view which employee names and current titles.
    c. A view which shows employee names, current titles, and skills.

12. Use GRANT to establish the following privileges:

    a. Allow user JOE read access on the PROJECT relation.
    b. Allow anyone who has access to the database to read the CURRENT_TITLE table.
    c. Allow DAVID to have all privileges on the EMPLOYEE relation.
    d. Allow JOE to add data to the PROJECT table.

13. Write COBOL programs to perform the following tasks:

    a. Accept an employee number and a project number to which that employee has been assigned. Check to see that the project number exists in the PROJECT relation.
    b. Accept data from the user to add a new row to the EMPLOYEE table.
    c. Accept a project number from the user and delete all information about that project from the PROJECT table and the ASSIGNMENT table.

FOR FURTHER READING

Early papers on SQL are:

Chamberlin, D.D., et al. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM Journal of Research and Development,* Vol. 20, No. 6, November 1976.

Chamberlin, D.D., and R.F. Boyce. "SEQUEL, A Structured English Query Language," *Proceedings of the ACM SIGMOD Workshop on Data Description, Access, and Control,* 1974.

For critical review of SQL, see:

Date, C.J. "Where SQL Falls Short," *Datamation,* Vol 33, No. 9, 1987, pp. 83-86.

Descriptions of more recent implementations may be found in:

SQL/Data System Terminal Users' Guide, IBM Corporation, 1982.
Oracle Users' Guide, Oracle Corporation, 1983.

For the formal SQL standards, see:

American National Standards Institute, "Database Language SQL, *Document ANSI X3*, 135-1986, 1986.

For more on specific implementations and a very readable presentation, see:

Ageloff, Roy, "A Primer on SQL", Times Mirror/Mosby College Publishing, St. Louis, 1988.

```
SELECT NAME, E_NUMBER
FROM DEPARTMENT
```

```
EMPLOYEE
NAME E_NUMBER

Kanter          1111
Yates           1112
Adams 1001
Baker           1002
Clarke          1003
Dexter          1004
Early           1005
```

(a)

```
SELECT DEPARTMENT
FROM EMPLOYEE
```

```
EMPLOYEE
DEPARTMENT

Finance
Accounting
Finance
Finance
Accounting
Finance
Accounting
```

(b)

FIGURE 7-2
Projection. (a) Projection of EMPLOYEE over Name and E-Number; (b) projection of EMPLOYEE over Department;

```
SELECT DISTINCT DEPARTMENT
FROM EMPLOYEE
```

EMPLOYEE
DEPARTMENT

Finance
Accounting

(c)

FIGURE 7-2
Projection. (c) another projection of EMPLOYEE over department. This time the DISTINCT keyword is used to remove duplicate tuples.

```
SELECT NAME, ACTUAL_COST - EXPECTED_COST
FROM PROJECT
```

```
NAME                          ACTUAL_COST - EXPECTED_COST

New billing system                     -9,000
Common stock issue               -1,000
Resolve bad debts                       500
New office lease                          0
Revise documentation             -2,900
Entertain new client                    3,000
New TV commerical                 2,000
```

(a)

```
SELECT NAME, ACTUAL_COST / EXPECTED_COST * 100.0
FROM PROJECT
```

```
NAME                          ACTUAL_COST / EXPECTED_COST * 100.0

New billing system                      10.00
Common stock issue                75.00
Resolve bad debts                      133.33
New office lease                       100.00
Revise documentation               3.33
Entertain new client                   250.00
New TV commerical                125.00
```

(b)

Figure 7-3. (a) Projection based on actual cost less expected cost. (b) Projection based on percent of actual to expected cost.

```
SELECT *
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
```

EMPLOYEE

NAME	E_NUMBER	DEPARTMENT
Kanter	1111	Finance
Adams	1001	Finance
Baker	1002	Finance
Dexter	1004	Finance

FIGURE 7-4

Selection from EMPLOYEE where DEPARTMENT = 'Finance'

```
SELECT NAME, E_NUMBER
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
```

```
EMPLOYEE
NAME          E_NUMBER

Kanter             1111
Adams         1001
Baker              1002
Dexter             1004
```

(b)

```
SELECT E_NUMBER, NAME
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
```

```
EMPLOYEE
E_NUMBER   NAME

1111          Kanter
1001          Adams
1002          Baker
1004          Dexter
```

(c)

Figure 7-5 Combining Projection and Selection. (a) Selection of employees in the Finance Department projected over NAME; (b) Selection of employees in the Finance Department, projected over NAME and E_NUMBER; (c) As in (b), but column order is interchanged.

```
SELECT *
FROM EMPLOYEE, TITLE
WHERE EMPLOYEE.E_NUMBER = TITLE.E_NUMBER
```

EMPLOYEE

NAME	E_NUMBER	DEPARTMENT	CURRENT_TITLE
Adams	1001	Finance	Senior consultant
Baker	1002	Finance	Senior consultant
Clarke	1003	Accounting	Senior consultant
Dexter	1004	Finance	Junior consultant
Early	1005	Accounting	Junior consultant

FIGURE 7-6
Join of EMPLOYEE and TITLE over E_NUMBER.
This is a natural join because the redundant attribute has been removed.

```
SELECT NAME
FROM EMPLOYEE, TITLE
WHERE EMPLOYEE.E_NUMBER = TITLE.E_NUMBER
AND CURRENT_TITLE = 'Senior Consultant'
```

```
NAME

Adams
Baker
Clarke
```

(a)

```
SELECT NAME, SKILL
FROM EMPLOYEE, TITLE, EXPERTISE
WHERE EMPLOYEE.E_NUMBER = TITLE.E_NUMBER
AND CURRENT_TITLE = 'Senior Consultant'
AND EXPERTISE.E_NUMBER = TITLE.E_NUMBER
```

```
NAME        SKILL

Adams       Stock market
Adams       Investments
Baker             Stock market
Clarke            Stock market
Clarke            Investments
```

(b)

FIGURE 7-7
Join: (a) Selection of senior consultants from TITLE joined with EMPLOYEE over E_NUMBER and projected over NAME; (b) As in (a), except also joined with EXPERTISE over E_NUMBER and projected over NAME and SKILL.

```
SELECT E_NUMBER
FROM EXPERTISE
WHERE SKILL IN ('Taxation', 'Management')
```

```
EXPERTISE
E_NUMBER

   1004
   1005
```

FIGURE 7-8.
Selection from EXPERTISE where SKILL is in the set of values (Taxation, Management, projected over E_NUMBER.

```
SELECT SKILL
FROM EXPERTISE
WHERE E_NUMBER IN
        (SELECT E_NUMBER
        FROM PRIOR-JOB
        WHERE PRIOR_TITLE = 'Research analyst')
```

```
EXPERTISE
SKILL

Stock market
Investments
Stock market
```

(a)

```
SELECT SKILL
FROM EXPERTISE
WHERE E_NUMBER NOT IN
        (SELECT E_NUMBER
        FROM PRIOR_JOB
        WHERE PRIOR_TITLE = 'Research analyst')
```

```
EXPERTISE
SKILL

Stock market
Investments
Taxation
Management
```

(b)

FIGURE 7-9
Selection with Subquery. (a) Selection from EXPERTISE where the prior job title is Research analyst; (b) Selection from EXPERTISE where prior job title is not Research analyst.

```
SELECT NAME
FROM PROJECT
WHERE MANAGER = 'Kanter'
AND ACTUAL_COST > EXPECTED_COST
```

PROJECT

NAME
Resolve Bad Debts

FIGURE 7-10. A More Complicated WHERE clause.
Selection from PROJECT where the manager is Kanter and actual cost exceeds expected cost.

```
PROGRAM OVER-COST/This program retrieves the names of projects for specfied project
        manager that are over expected cost/
BEGIN
Reset Project File
Display "Please enter Manager: name:"
Accept Mgr-Name
While not EOF DO
Begin Loop Read Project File: Project, Number Manager, Actual-Cost,
                        Expected-Cost
If Mgr-Name = Manager and Actual-Cost > expected-Cost
                Then
                        Display project
                End Loop
        END PROGRAM
```

Figure 7-11
A Pseudocode Program to Achieve the Result Shown in Figure 7-10.

```
SELECT NAME
FROM PROJECT
WHERE ACTUAL_COST > EXPECTED_COST
AND ((MANAGER = 'Kanter') OR (MANAGER = 'Baker'))
```

```
PROJECT
NAME

Resolve bad debts
New TV commercial
```

(a)

```
SELECT NAME
FROM PROJECT
WHERE EXPECTED_COST BETWEEN 4000 AND 8000.
```

```
PROJECT
NAME

Common stock issue
New office lease
New TV commercial
```

(b)

```
SELECT NAME
FROM PROJECT
WHERE (ACTUAL_COST - EXPECTED_COST) / EXPECTED_COST * 100.0 > 20.0
```

```
PROJECT
NAME

Resolve bad debts
Entertain new client
New TV commercial
```

(c)

FIGURE 7-12
(a) Selection from PROJECT Where Actual Cost > Expected Cost and Manager = 'Kanter' or 'Baker,' Projected Over Name;
(b) Projects with Expected Cost BETWEEN 4,000 and 8,000;
(c) Projects that are at least 20% over budget

```
SELECT NAME
FROM EMPLOYEE
WHERE DEPARTMENT = 'Finance'
ORDER BY NAME
```

EMPLOYEE NAME
Adams
Baker
Dexter
Kanter

Figure 7-13
Sorting Results. Use of ORDER BY to alphabetize finance employees.

```
SELECT DEPARTMENT, NAME, E_NUMBER
FROM EMPLOYEE
ORDER BY DEPARTMENT, NAME

EMPLOYEE
DEPARTMENT    NAME                E_NUMBER

Accounting    Clarke              1003
Accounting    Early               1005
Accounting    Yates               1112
Finance       Adams               1001
Finance       Baker               1002
Finance       Dexter              1004
Finance       Kanter              1111
```

Figure 7-14
Output alphabatized by Department and Name within Department.

```
SELECT *
FROM PROJECT
WHERE EXPECTED_COST >
        (SELECT AVG (EXPECTED_COST)
         FROM PROJECT)
```

```
PROJECT
NAME                 P_NUMBER   MANAGER   ACTUAL_COST   EXPECTED_COST

New billing system  23760   Yates      1000       10000
New office lease        26511    Yates      5000       5000
New TV commercial       85005   Baker      10000      8000
```

(a)

```
SELECT *
FROM PROJECT
WHERE ACTUAL_COST =
        (SELECT MAX (ACTUAL_COST)
         FROM PROJECT)
```

```
PROJECT
NAME                 P_NUMBER       MANAGER   ACTUAL_COST   EXPECTED_COST

New TV commercial    85005          Baker      10000      8000
```

(b)

```
SELECT *
FROM PROJECT
WHERE ACTUAL_COST =
        (SELECT MIN (ACTUAL_COST)
         FROM PROJECT)
```

```
PROJECT
NAME                 P_NUMBER   MANAGER   ACTUAL_COST   EXPECTED_COST

Resolve bad debts  26713     Kanter      2000       1500
```

(c)

Figure 7-15.
Aggregate Functions. (a) Comparison of expected cost to average expected cost to find those projects with a larger than average budget; (b) the project with the largest budget; (c) the project with the smallest budget.

```
SELECT MANAGER, SUM (ACTUAL_COST)
FROM PROJECT
GROUP BY MANAGER

MANAGER                              SUM (ACTUAL_COST)

Yates                                      11,000
Baker                                      13,000
Kanter                                      2,100
```

(a)

```
SELECT MANAGER, SUM (ACTUAL-COST)
FROM PROJECT
GROUP BY MANAGER
ORDER BY MANAGER

MANAGER                              SUM (ACTUAL_COST)

Baker                                13,000
Kanter                                2,100
Yates                                11,000
```

(b)

```
SELECT MANAGER, SUM (ACTUAL-COST)
FROM PROJECT
GROUP BY MANAGER
HAVING SUM (ACTUAL-COST) > 5000
ORDER BY MANGER

MANAGER                              SUM (ACTUAL_COST)

Baker                                      13,000
Yates                                      11,000
```

(c)

Figure 7-16. (a) Use of the GROUP BY clause to sum actual costs of projects managed by each person.(b) Use of ORDER BY to sort the results in Figure 7-16a. (c) HAVING has been used to restrict the output to mangers spending more than 5000. ORDER BY was used to sort results.

```
SELECT MANGER, SUM (ACTUAL_COST - EXPECTED_COST)
FROM PROJECT
WHERE ACTUAL_COST > EXPECTED_COST
GROUP BY MANAGER

MANAGER                          SUM (ACTUAL COST)

Kanter                             500
Yates                            3,000
Baker                            2,000
```

(d)

Figure 7-16. (d) Use of WHERE to restrict results to rows in which actual cost exceeds expected cost, and GROUP BY to organize results by manager.