6 CHAPTER

Relational Database Management Systems and SQL

Chapter Objectives

In this chapter you will learn the following:

- The history of relational database systems and SQL
- How the three-level architecture is implemented in relational database management systems
- How to create and modify a conceptual-level database structure using SQL DDL
- How to retrieve and update data in a relational database using SQL DML
- How to enforce constraints in relational databases

6.0 Chapter Objectives

6.1 Brief History of SQL in Relational Database Systems

6.2 Architecture of a Relational Database Management System

6.3 Defining the Database: SQL DDL

6.3.1 CREATE TABLE

6.3.1.1 Data Types

6.3.1.2 Column and Table Constraints

6.3.2 CREATE INDEX

6.3.3 ALTER TABLE, RENAME TABLE

6.3.4 DROP Statements

6.4 Manipulating the Database: SQL DML

6.4.1 Introduction to the SELECT Statement

6.4.2 SELECT Using Multiple Tables

6.4.3 SELECT with Other Operators

6.4.4 Operators for Updating: UPDATE, INSERT, DELETE

6.5 Active Databases

6.5.1 Enabling and Disabling Constraints

6.5.2 SQL Triggers

6.6 Using COMMIT and ROLLBACK Statements

6.7 SQL Programming

6.7.1 Embedded SQL

6.7.2 ODBC and JDBC

6.7.3 SQL PSM

6.8 Creating and Using Views

6.9 The System Catalog

6.10 Chapter Summary

Exercises

Lab Exercises

1. Exploring the Oracle Database for University Example (provided on CD)
2. Creating and Using a Simple Database in Oracle

SAMPLE PROJECT

Steps 6.1–6.6: Creating and Using a Relational Database for The Art Gallery

STUDENT PROJECTS

Steps 6.1–6.6: Creating and Using a Relational Database for the Student Projects

- How to terminate relational transactions
- How SQL is used in a programming environment
- How to create relational views
- When and how to perform operations on relational views
- The structure and functions of a relational database system catalog
- The functions of the various components of a relational database management system

6.1 Brief History of SQL in Relational Database Systems

As described in Chapter 4, the relational model was first proposed by E. F. Codd in 1970. D. D. Chamberlin and others at the IBM San Jose Research Laboratory developed a language now called SQL, or Structured Query Language, as a data sublanguage for the relational model. Originally spelled SEQUEL, the language was presented in a series of papers starting in 1974, and it was used in a prototype relational system called System R, which was developed by IBM in the late 1970s. Other early prototype relational database management systems included INGRES, which was developed at the University of California at Berkeley, and the Peterlee Relational Test Vehicle, developed at the IBM UK Scientific Laboratory. System R was evaluated and refined over a period of several years, and it became the basis for IBM's first commercially available relational database management system, SQL/DS, which was announced in 1981. Another early commercial database management system, Oracle, was developed in the late 1970s using SQL as its language. IBM's DB2, also using SQL as its language, was released in 1983. Microsoft SQL Server, MySQL, Informix, Sybase, dBase, Paradox, r: Base, FoxPro, and hundreds of other relational database management systems have incorporated SQL.

Both the American National Standards Institute (ANSI) and the International Standards Organization (ISO) adopted SQL as a standard language for relational databases and published specifications for the SQL language in 1986. This standard is usually called SQL1. A minor revision, called SQL-89, was published three years later. A major revision, SQL2, was adopted by both ANSI and ISO in 1992. The first parts of the SQL3 standard, referred to as SQL:1999, were published in 1999. Major new features included object-oriented data management capabilities and user-defined data types. Most vendors of relational database management systems use their own extensions of the language, creating a variety of dialects around the standard.

SQL has a complete data definition language (DDL) and data manipulation language (DML) described in this chapter, and an authorization language, described in Chapter 9. Readers should note that different implementations of SQL vary slightly from the standard syntax presented here, but the basic notions are the same.

6.2 Architecture of a Relational Database Management System

Relational database management systems support the standard three-level architecture for databases described in Section 2.6. As shown in Figure 6.1, relational databases provide both logical and physical data independence because they separate the external, conceptual, and internal levels. The conceptual level, which corresponds to the logical level for relational databases, consists of base tables that are physically stored. These tables are created by the database administrator using a CREATE TABLE command, as described in Section 6.3. A base table can have any number of indexes, created by the DBA using the CREATE INDEX command. An index is used to speed up retrieval of records based on the value in one or more columns. An index lists the values that exist for the indexed column(s), and the location of the records that have those values. Most relational database management systems use B trees or B+ trees for indexes. (See Appendix A.) On the physical level, the base tables are represented, along with their indexes, in files. The physical representation of the tables may not correspond exactly to our notion of a base table as a two-dimensional object consisting of rows and columns. However, the rows of the table do correspond to physically stored records, although their order and other

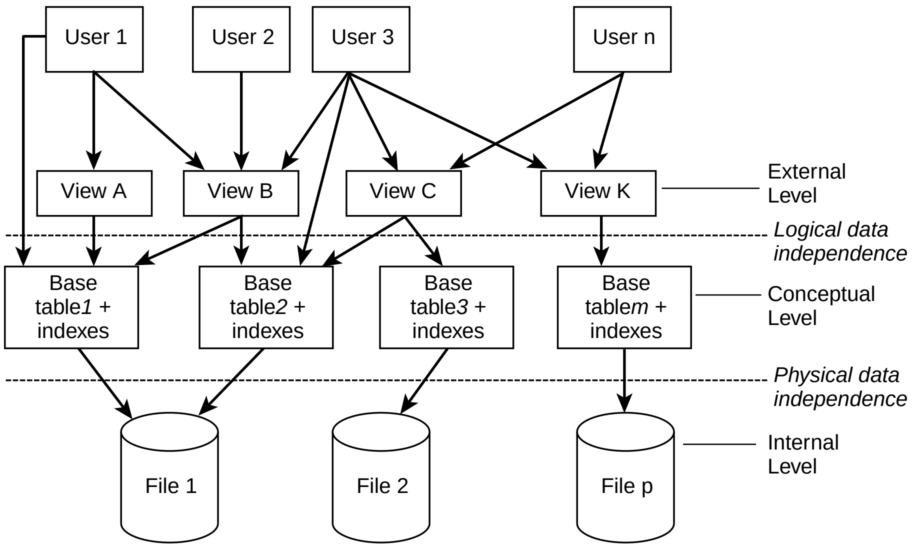


FIGURE 6.1
Three level architecture for relational databases

details of storage may be different from our concept of them. The database management system, not the operating system, controls the internal structure of both the data files and the indexes. The user is generally unaware of what indexes exist, and has no control over which index will be used in locating a record. Once the base tables have been created, the DBA can create "views" for users, using the CREATE VIEW command, described in Section 6.8. A view may be a subset of a single base table, or it may be created by combining base tables. Views are "virtual tables," not permanently stored, but created when the user needs to access them. Users are unaware of the fact that their views are not physically stored in table form. In a relational system, the word "view" means a single virtual table. This is not exactly the same as our term "external view," which means the database as it appears to a particular user. In our terminology, an external view may consist of several base tables and/or views.

One of the most useful features of a relational database is that it permits dynamic database definition. The DBA, and users he or she authorizes to do so, can create new tables, add columns to old ones, create new indexes, define views, and drop any of these objects at any time. By contrast, many other systems require that the entire database structure be defined at creation time, and that the entire system be halted and reloaded when any structural changes are made. The flexibility of relational databases encourages users to experiment with various structures and allows the system to be modified to meet their changing needs. This enables the DBA to ensure that the database is a useful model of the enterprise throughout its life cycle.

6.3 Defining the Database: SQL DDL

The most important SQL Data Definition Language (DDL) commands are the following:

CREATE TABLE
CREATE INDEX
ALTER TABLE
RENAME TABLE
DROP TABLE
DROP INDEX

These statements are used to create, change, and destroy the logical structures that make up the conceptual model. These commands can be used at any time to make changes to the database structure. Additional commands are available to specify physical details of storage, but we will not discuss them here, since they are specific to the system.

We will apply these commands to the following example, which we have used in previous chapters:

```
Student (stuId, lastName, firstName, major, credits)
Faculty (facId, name, department, rank)
Class (classNumber, facId, schedule, room)
Enroll (classNumber, stuId, grade)
```

6.3.1 Create Table

This command is used to create the base tables that form the heart of a relational database. Since it can be used at any time during the lifecycle of the system, the database developer can start with a small number of tables and add to them as additional applications are planned and developed. A base table is fairly close to the abstract notion of a relational table. It consists of one or more **column headings,** which give the **column name** and **data type,** and zero or more **data rows,** which contain one data value of the specified data type for each of the columns. As in the abstract relational model, the rows are considered unordered. However, the columns are ordered left-to-right, to match the order of column definitions in the CREATE TABLE command. The form of the command is:

```
CREATE TABLE base-table-name (colname datatype [column constraints]
[,colname datetype [column constraints - NULL/NOT NULL, DEFAULT . . . ,
  UNIQUE, CHECK . . . , PRIMARY KEY . . .]]
. . .
[table constraints - PRIMARY KEY . . . , FOREIGN KEY . . . , UNIQUE
. . . , CHECK . . .]
[storage specifications]);
```

Here, *base-table-name* is a user-supplied name for the table. No SQL keywords may be used, and the table name must be unique within the database. For each column, the user must specify a name that is unique within the table, and a data type. The optional storage specifications section of the CREATE TABLE command allows the DBA to name the tablespace where the table will be stored. If the tablespace is not specified, the database management system will create a default space for the table. Those who wish to can ignore system details and those who desire more control can be very specific about storage areas.

Figure 6.2 shows the commands to create the base tables for a database for the University example.

6.3.1.1 Data Types

Built-in data types include various numeric types, fixed-length and varying-length character strings, bit strings, and user-defined types. The available data types vary from DBMS to DBMS. For example, the most common types in Oracle are CHAR(N), VARCHAR2(N), NUMBER(N,D), DATE, and BLOB (binary large object). In DB2, types include SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, REAL, DOUBLE, CHAR(N), VARCHAR(N), LONG VARCHAR, CLOB, GRAPHIC, DBCLOB, BLOB, DATE, TIME, and TIMESTAMP. Microsoft SQL Server types include NUMERIC, BINARY, CHAR, VARCHAR DATETIME, MONEY, IMAGE, and others. Microsoft Access supports several types of NUMBER, as well as TEXT, MEMO, DATE/TIME, CURRENCY, YES/NO, and others. In addition, some systems, such as Oracle, allow users to create new domains, built on existing data types. Rather than using one of the built-in data types, users can specify domains in advance, and they can include a check condition for the domain. SQL:1999 allows

FIGURE 6.2
SQL DDL statements to create Oracle tables for the University example.

```
CREATE TABLE Student           (
stuId                          CHAR(6),
lastName                       CHAR(20) NOT NULL,
firstName                      CHAR(20) NOT NULL,
major                          (CHAR(10),
credits                        SMALLINT DEFAULT 0,
CONSTRAINT Student_stuId_pk PRIMARY KEY (stuId)),
CONSTRAINT Student_credits_cc CHECK ((CREDITS>=0) AND (credits < 150));
CREATE TABLE Faculty           (
facId                          CHAR(6),
name                           CHAR(20) NOT NULL,
department                     CHAR(20) NOT NULL,
rank                           CHAR(10),
CONSTRAINT Faculty_facId_pk PRIMARY KEY (facId));
CREATE TABLE Class             (
classNumber                    CHAR(8),
facId                          CHAR(6) NOT NULL,
schedule                       CHAR(8),
room                           CHAR(6),
CONSTRAINT Class_classNumber_pk PRIMARY KEY (classNumber),
CONSTRAINT Class_facId_fk FOREIGN KEY (facId) REFERENCES Faculty (facId) ON DELETE
   NO ACTION);
CREATE TABLE Enroll            (
classNumber                    CHAR(8),
stuId                          CHAR(6),
grade                          CHAR(2),
CONSTRAINT Enroll_classNumber_stuId_pk PRIMARY KEY (classNumber, stuId),
CONSTRAINT Enroll_classNumber_fk FOREIGN KEY (classNumber) REFERENCES Class
   (classNumber) ON DELETE NO ACTION,
CONSTRAINT Enroll_stuId_fk FOREIGN KEY (stuId) REFERENCES Student (stuId) ON DELETE CASCADE);
```

the creation of new **distinct** data types using one of the previously defined types as the source type. For example, we could write:

```
CREATE DOMAIN creditValues INTEGER
DEFAULT 0
CHECK (VALUE >=0 AND VALUE <150);
```

Once a domain has been created, we can use it as a data type for attributes. For example, when we create the Student table, for the specification of `credits` we could then write:

```
credits creditValues, . . .
```

in place of,

```
credits SMALLINT DEFAULT 0,
. . .
CONSTRAINT Student_credits_cc CHECK ((credits>=0) AND (credits < 150);
```

However, when we create distinct types, SQL:1999 does not allow us to compare their values with values of other attributes having the same underlying source type. For example, if we use the `creditValues` domain for `credits`, we cannot compare `credits` with another attribute whose type is also SMALLINT—for example, with `age`, if we had stored that attribute. We cannot use the built-in SQL functions such as COUNT, AVERAGE, SUM, MAX, or MIN on distinct types, although we can write our own definitions of functions for the new types.

6.3.1.2 Column and Table Constraints

The database management system has facilities to enforce data correctness, which the DBA should make use of when creating tables. Recall from Section 4.4 that the relational model uses integrity constraints to protect the correctness of the database, allowing only legal instances to be created. These constraints protect the system from data entry errors that would create inconsistent data. Although the table name, column names, and data types are the only parts required in a CREATE TABLE command, optional constraints can and should be added, both at the column level and at the table level.

The column constraints include options to specify NULL/NOT NULL, UNIQUE, PRIMARY KEY, CHECK and DEFAULT for any column, immediately after the specification of the column name and data type. If we do not specify NOT NULL, the system will allow the column to have null values, meaning the user can insert records that have no values for those fields. When a null value appears in a field of a record, the system is able to distinguish it from a blank or zero value, and treats it differently in computations and logical comparisons. It is desirable to be able to insert null values in certain situations; for example, when a college student has not yet declared a major we might want to set the `major` field to null. However, the use of null values can create complications, especially in operations such as joins, so we should use NOT NULL when it is appropriate. We can also specify a default value for a column, if we wish to do so. Every record that is inserted without a value for that field will then be given the default value automatically. We can optionally specify that a given field is to have unique values by writing the UNIQUE constraint. In that case, the system will reject the insertion of a new record that has the same value in that field as a record that is already in the database. If the primary key is not composite, it is also possible to specify PRIMARY KEY as a column constraint, simply by adding the words PRIMARY KEY after the data type for the column. Clearly we cannot allow duplicate values for the primary key. We also disallow null values, since we could not distinguish between two different records if they both had null key values, so the specification of PRIMARY KEY in SQL carries an implicit NOT NULL constraint as well as a UNIQUE constraint. However, we may wish to ensure uniqueness for candidate keys as well, and we should specify UNIQUE for them when we create the table. The system automatically checks each record we try to insert to ensure that data items for columns that have been described as unique do not have values that duplicate any other data items in the database for those columns. If a duplication might occur, it will reject the insertion. It is also desirable to specify a NOT NULL constraint for candidate keys, when it is possible to ensure that values for these columns will always be available. The CHECK constraint can be used to verify that values provided for attributes are appropriate. For example, we could write:

```
credits SMALLINT DEFAULT 0 CHECK ((credits>=0) AND (credits < 150)),
```

Table constraints, which appear after all the columns have been declared, can include the specification of a primary key, foreign keys, uniqueness, checks, and general constraints that can be expressed as conditions to be checked. If the primary key is a composite, it must be identified using a table constraint rather than a column constraint, although even a primary key consisting of a single column can be identified as a table constraint. The PRIMARY KEY constraint enforces the uniqueness and not null constraints for the column(s) identified as the primary key. The FOREIGN

KEY constraint requires that we identity the referenced table where the column or column combination is a primary key. The SQL standard allows us to specify what is to be done with records containing the foreign key values when the records they relate to are updated or deleted in their home table. For the University example, what should happen to a Class record when the record of faculty member assigned to teach the class is deleted or the `facId` of the Faculty record is updated? For the deletion case, the DBMS could automatically:

- Delete all Class records for that faculty member, an action performed when we specify ON DELETE CASCADE in the foreign key specification in SQL.
- Set the `facId` in the Class record to a null value, an action performed when we write ON DELETE SET NULL in SQL.
- Set the `facId` to some default value such as F999 in the Class table, an action performed when we write ON DELETE SET DEFAULT in SQL. (This choice requires that we use the DEFAULT column constraint for this column prior to the foreign key specification.)
- Not allow the deletion of a Faculty record if there is a Class record that refers to it, an action performed when we specify ON DELETE NO ACTION in SQL.

The same actions, with similar meanings, can be specified in an ON UPDATE clause; that is,

```
ON UPDATE CASCADE/SET NULL/SET DEFAULT/NO ACTION
```

For both deletion and update, the default is NO ACTION, essentially disallowing changes to a record in a home relation that would cause inconsistency with records that refer to it. As shown in Figure 6.2, for the Class table we have chosen the ON UPDATE CASCADE. Also note the choices we made for the Enroll table, for changes made to both `classNumber` and `stuId`.

The table uniqueness constraint mechanism can be used to specify that the values in a combination of columns must be unique. For example, to ensure that no two classes have exactly the same schedule and room, we would write:

```
CONSTRAINT Class_schedule_room_uk UNIQUE (schedule, room)
```

Recall from Section 4.4 that the uniqueness constraint allows us to specify candidate keys. The above constraint says that `{schedule, room}` is a candidate key for Class. We could also specify that `{facId, schedule}` is a candidate key by,

```
CONSTRAINT Class_facId_schedule_uk UNIQUE (facId, schedule)
```

since a faculty member cannot teach two classes with exactly the same schedule.

Constraints, whether column or table level, can optionally be given a name, as illustrated in the examples. If we do not name them, the system will generate a unique constraint name for each constraint. The advantage of naming constraints is that we can then refer to them easily. There are SQL commands to allow us to disable, enable, alter, or drop constraints at will, provided we know their names. It is good practice to use a consistent pattern in naming constraints. The pattern illustrated here is the table-name, column name(s) and an abbreviation for the constraint type (pk, fk, nn, uk, cc), separated by underscores.

6.3.2 Create Index

We can optionally create indexes for tables to facilitate fast retrieval of records with specific values in a column. An index keeps track of what values exist for the indexed column, and which records have those values. For example, if we have an index on the `lastName` column of the Student table, and we write a query asking for all students with last name of Smith, the system will not have to scan all Student records to pick out the desired ones. Instead, it will read the index, which will point it to the records with the desired name. A table can have any number of indexes, which are stored as B-trees or B+ trees in separate index files, usually close to the tables they index. (See Appendix A for a description of tree indexes.) Indexes can be created on single fields or combinations of fields. However, since indexes must be updated by the system every time the underlying tables are updated, additional overhead is required. Aside from choosing which indexes will exist, users have no control over the use or maintenance of indexes. The system chooses which, if any, index to use in searching for records. Indexes are not part of the SQL standard, but most DBMSs support their creation. The command for creating an index is:

```
CREATE [UNIQUE] INDEX indexname ON basetablename (colname [order]
[,colname [order]] . . .) [CLUSTER] ;
```

If the UNIQUE specification is used, uniqueness of the indexed field or combination of fields will be enforced by the system. Although indexes can be created at any time, we may have a problem if we try to create a unique index after the table has records stored in it, because the values stored for the indexed field or fields may already contain duplicates. In this case, the system will not allow the unique index to be created. To create the index on lastName for the Student table we would write:

```
CREATE INDEX Student_lastName ON STUDENT (lastName);
```

The name of the index should be chosen to indicate the table and the field or fields used in the index. Any number of columns, regardless of where they appear on the table, may be used in an index. The first column named determines major order, the second gives minor order, and so on. For each column, we may specify that the order is ascending, ASC, or descending, DESC. If we choose not to specify order, ASC is the default. If we write,

```
CREATE INDEX Faculty_department_name ON Faculty (department ASC,
name ASC);
```

then an index file called Faculty_Department_Name will be created for the `Faculty` table. Entries will be in alphabetical order by department. Within each department, entries will be in alphabetical order by faculty name.

Some DBMSs allow an optional CLUSTER specification for only one index for each table. If we use this option, the system will store records with the same values for the indexed field(s) close together physically, on the same page or adjacent pages if possible. If we create a clustered index for the field(s) used most often for retrieval, we can substantially improve performance for those applications needing that particular order of retrieval, since we will be minimizing seek time and read time. However, it is the system, not the user, that chooses to use a particular index, even a clustered one, for data retrieval.

Oracle automatically creates an index on the primary key of each table that is created. The user should create additional indexes on any field(s) that are often used in queries, to speed up execution of those queries. Foreign key fields, which are often used in joins, are good candidates for indexing.

6.3.3 ALTER TABLE, RENAME TABLE

Once a table has been created, users might find that it more useful if it contained an additional data item, did not have a particular column, or had different constraints. Here, the dynamic nature of a relational database structure makes it possible to change existing base tables. For example, to add a new column on the right of the table, we use a command of the form:

```
ALTER TABLE basetablename ADD columnname datatype;
```

Notice we cannot use the NULL specification for the column. An ALTER TABLE ..ADD command causes the new field to be added to all records already stored in the table, and null values to be assigned to that field in all existing records. Newly inserted records, of course, will have the additional field, but we are not permitted to specify no nulls even for them.

Suppose we want to add a new column, `cTitle`, to our `Class` table. We can do so by writing

```
ALTER TABLE Class ADD cTitle CHAR(30);
```

The schema of the `Class` table would then be:

```
Class(classNumber,facId,schedule,room,cTitle)
```

All old `Class` records would now have null values for `cTitle`, but we could provide a title for any new `Class` records we insert, and update old `Class` records by adding titles to them. We can also drop columns from existing tables by the command:

```
ALTER TABLE basetablename DROP COLUMN columnname;
```

To drop the `cTitle` column and return to our original structure for the Class table, we would write:

```
ALTER TABLE Class DROP COLUMN cTitle;
```

If we want to add, drop, or change a constraint, we can use the same ALTER TABLE command. For example, if we created the Class table and neglected to make `facId` a foreign key in Class, we could add the constraint at any time by writing:

```
ALTER TABLE Class ADD CONSTRAINT Class_facId_fk FOREIGN KEY (facId)
REFERENCES Faculty (facId)ON DELETE NO ACTION);
```

We could drop an existing named constraint using the ALTER TABLE command. For example, to drop the check condition on the `credits` attribute of Student that we created earlier, we could write:

```
ALTER TABLE Student DROP CONSTRAINT Student_credits_cc;
```

We can change the name of an existing table easily by the command:

```
RENAME TABLE old-table-name TO new-table-name;
```

6.3.4 DROP Statements

Tables can be dropped at any time by the SQL command:

```
DROP TABLE basetablename;
```

When this statement is executed, the table itself and all records contained in it are removed. In addition, all indexes and, as we will see later, all views that depend on it are dropped. Naturally, the DBA confers with potential users of the table before taking such a drastic step. Any existing index can be destroyed by the command:

```
DROP INDEX indexname;
```

The effect of this change may or may not be seen in performance. Recall that users cannot specify when the system is to use an index for data retrieval. Therefore, it is possible that an index exists that is never actually used, and its destruction would have no affect on performance. However, the loss of an efficient index that is used by the system for many retrievals would certainly affect performance. When an index is dropped, any access plans for applications that depend on it are marked as invalid. When an application calls them, a new access plan is devised to replace the old one.

6.4 Manipulating the Database: SQL DML

SQL's query language is **declarative,** also called **non-procedural,** which means that it allows us to specify what data is to be retrieved without giving the procedures for retrieving it. It can be used as an interactive language for queries, embedded in a host programming language, or as a complete language in itself for computations using SQL/PSM (Persistent Stored Modules).

The SQL DML statements are:

SELECT
UPDATE
INSERT
DELETE

6.4.1 Introduction to the SELECT Statement

The SELECT statement is used for retrieval of data. It is a powerful command, performing the equivalent of relational algebra's SELECT, PROJECT, and JOIN, as well as other functions, in a single, simple statement. The general form of SELECT is,

```
SELECT    [DISTINCT] col-name [AS newname], [,col-name..] . . .
FROM      table-name [alias] [,table-name] . . .
[WHERE    predicate]
[GROUP BY col-name [,col-name] . . . [HAVING predicate]
```

or,

```
[ORDER BY col-name [,col-name] . . .];
```

The result is a table that may have duplicate rows. Since duplicates are allowed in such a table, it is not a relation in the strict sense, but is referred to as a **multi-set** or a **bag.** As indicated by the absence of square brackets, the SELECT and the FROM clauses are required, but not the WHERE or the other clauses. The many variations of this statement will be illustrated by the examples that follow, using the `Student, Faculty, Class,` and/or `Enroll` tables as they appear in Figure 6.3

- Example 1. Simple Retrieval with Condition

  Question: Get names, IDs and number of credits of all Math majors.

  Solution: The information requested appears on the `Student` table. From that table we select only the rows that have a value of 'Math' for `major`. For those rows, we display only the `lastName, firstName, stuId,` and `credits` columns. Notice we are doing the equivalent of relational algebra's SELECT (in finding the rows) and PROJECT (in displaying only certain columns). We are also rearranging the columns.

  SQL Query:

  ```
  SELECT   lastName, firstName, stuId, credits
  FROM     Student
  WHERE    major = 'Math';
  ```

  Result:

lastName	firstName	stuId	credits
Jones	Mary	S1015	42
Chin	Ann	S1002	36
McCarthy	Owen	S1013	9

Notice that the result of the query is a table or a multi-set.

Student

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edward	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	CSC	15

Faculty

facId	name	department	rank
F101	Adams	Art	Professor
F105	Tanaka	CSC	Instructor
F110	Byrne	Math	Assistant
F115	Smith	History	Associate
F221	Smith	CSC	Professor

Class

classNumber	facId	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TuThF10	M110
CSC203A	F105	MThF12	M110
HST205A	F115	MWF11	H221
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

Enroll

stuId	classNumber	grade
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

FIGURE 6.3
The University Database (Same as Figure 1.1)

- Example 2. Use of Asterisk Notation for "all columns"

  Question: Get all information about CSC Faculty.

  Solution: We want the entire `Faculty` record of any faculty member whose department is 'CSC'. Since many SQL retrievals require all columns of a single table, there is a short way of expressing "all columns," namely by using an asterisk in place of the column names in the SELECT line.

SQL Query:

```
SELECT    *
FROM      Faculty
WHERE     department = 'CSC';
```

Result:

facId	name	department	rank
F105	Tanaka	CSC	Instructor
F221	Smith	CSC	Professor

Users who access a relational database through a host language are usually advised to avoid using the asterisk notation. The danger is that an additional column might be added to a table after a program was written. The program will then retrieve the value of that new column with each record and will not have a matching program variable for the value, causing a loss of correspondence between database variables and program variables. It is safer to write the query as:

```
SELECT    facId, name, department, rank
FROM      Faculty
WHERE     department = 'CSC';
```

- Example 3. Retrieval without Condition, Use of "Distinct," Use of Qualified Names

  Question: Get the course number of all courses in which students are enrolled.

  Solution: We go to the `Enroll` table rather than the `Class` table, because it is possible there is a `Class` record for a planned class in which no one is enrolled. From the `Enroll` table, we could ask for a list of all the `classNumber` values, as follows.

  SQL Query:

```
SELECT    classNumber
FROM      Enroll;
```

Result:

```
classNumber
ART103A
CSC201A
CSC201A
ART103A
ART103A
MTHlOlB
HST205A
MTH103C
MTH103C
```

Since we did not need a predicate, we did not use the WHERE line. Notice that there are several duplicates in our result; it is a multi-set, not a true relation. Unlike the relational algebra PROJECT, the SQL SELECT does not eliminate duplicates when it "projects" over columns. To eliminate the duplicates, we need to use the DISTINCT option in the SELECT line. If we write,

```
SELECT DISTINCT classNumber
FROM Enroll;
```

The result would be:

```
classNumber
ART103A
CSC201A
MTH101B
HST205A
MTH103C
```

In any retrieval, especially if there is a possibility of confusion because the same column name appears on two different tables, specify *tablename.colname.* In this example, we could have written:

```
SELECT      DISTINCT Enroll.classNumber
FROM
Enroll;
```

Here, it is not necessary to use the qualified name, since the FROM line tells the system to use the `Enroll` table, and column names are always unique within a table. However, it is never wrong to use a qualified name, and it is sometimes necessary to do so when two or more tables appear in the FROM line.

- Example 4: Retrieving an Entire Table

  Question: Get all information about all students.

  Solution: Because we want all columns of the `Student` table, we use the asterisk notation. Because we want all the records in the table, we omit the WHERE line.

  SQL Query:

  ```
  SELECT    *
  FROM      Student;
  ```

  Result: The result is the entire `Student` table.

- Example 5. Use of "ORDER BY" and AS

  Question: Get names and IDs of all `Faculty` members, arranged in alphabetical order by name. Call the resulting columns FacultyName and FacultyNumber.

  Solution: The ORDER BY option in the SQL SELECT allows us to order the retrieved records in ascending (ASC—the default) or descending (DESC) order on any field or combination of fields, regardless of whether that field appears in the results. If we order by more than one field, the one named first determines major order, the next minor order, and so on.

  SQL Query:

  ```
  SELECT    name AS FacultyName, facId AS
            FacultyNumber
  FROM      Faculty
  ORDER BY  name;
  ```

  Result:

FacultyName	FacultyNumber
Adams	F101
Byrne	F110
Smith	F202
Smith	F221
Tanaka	F105

The column headings are changed to the ones specified in the AS clause. We can rename any column or columns for display in this way. Note the duplicate name of 'Smith'. Since we did not specify minor order, the system will arrange these two rows in any order it chooses. We could break the "tie" by giving a minor order, as follows:

```
SELECT    name AS FacultyName, facId AS
          FacultyNumber
FROM      Faculty
ORDER BY  name, department;
```

Now the Smith records will be reversed, since F221 is assigned to CSC, which is alphabetically before History. Note also that the field that determines ordering need not be one of the ones displayed.

- Example 6. Use of Multiple Conditions

  Question: Get names of all math majors who have more than 30 credits.

  Solution: From the `Student` table, we choose those rows where the major is 'Math' and the number of credits is greater than 30. We express these two conditions by connecting them with 'AND.' We display only the `lastName` and `firstName`.

  SQL Query:

```
SELECT    lastName, firstName
FROM      Student
WHERE     major = 'Math'
          AND credits > 30;
```

  Result:

lastName	firstName
Jones	Mary
Chin	Ann

The predicate can be as complex as necessary by using the standard comparison operators =, <>, <, <=, >, >= and the standard logical operators AND, OR and NOT, with parentheses, if needed or desired, to show order of evaluation.

6.4.2 SELECT Using Multiple Tables

- Example 7. Natural Join

  Question: Find IDs and names of all students taking ART103A.

  Solution: This question requires the use of two tables. We first look in the `Enroll` table for records where the `classNumber` is 'ART103A.' We then look up the `Student` table for records with matching `stuId` values, and join those records into a new table. From this table, we find the `lastName` and `firstName`. This is similar to the JOIN operation in relational algebra. SQL allows us to do a natural join, as described in Section 4.6.2, by naming the tables involved and expressing in the predicate the condition that the records should match on the common field.

  SQL Query:

```
SELECT    Enroll.stuId, lastName, firstName
FROM      Student, Enroll
WHERE     classNumber = 'ART103A'
          AND Enroll.stuId = Student.stuId;
```

  Result:

```
stuId     lastName    firstName
-------------------------------
Sl00l     Smith       Tom
Sl0l0     Burns       Edward
S1002     Chin        Ann
```

Notice that we used the qualified name for `stuId` in the SELECT line. We could have written `Student.stuId` instead of `Enroll.stuId`, but we needed to use one of the table names, because `stuId` appears on both of the tables in the FROM line. We did not need to use the qualified name for `classNumber` because it does not appear on the `Student` table. The fact that it appears on the `Class` table is irrelevant, as that table is not mentioned in the FROM line. Of course, we had to write both qualified names for `stuId` in the WHERE line.

Why is the condition "`Enroll.stuId=Student.stuId`" necessary? The answer is that it is essential. When a relational database system performs a join, it acts as if it first forms a Cartesian product, as described in Section 4.6.2, so an intermediate table containing the combinations of all records from the `Student` table with the records of the `Enroll` table is (theoretically) formed. Even if the system restricts itself to records in `Enroll` that satisfy the condition "`classNumber`='ART103A' ", the intermediate table numbers 6*3 or18 records. For example, one of those intermediate records is:

```
S1015  Jones  Mary  Math  42  ART103A  S1001  A
```

We are not interested in this record, since this student is not one of the people in the ART103A class. Therefore, we add the condition that the `stuId` values must be equal. This reduces the intermediate table to three records.

- Example 8. Natural Join with Ordering

  Question: Find `stuId` and `grade` of all students taking any course taught by the `Faculty` member whose `facId` is F110. Arrange in order by `stuId`.

  Solution: We need to look at the `Class` table to find the `classNumber` of all courses taught by F110. We then look at the `Enroll` table for records with matching `classNumber` values, and get the join of the tables. From this we find the corresponding `stuId` and `grade`. Because we are using two tables, we will write this as a join.

  SQL Query:

```
SELECT     stuId,grade
FROM       Class,Enroll
WHERE      facId = 'F110' AND Class.classNumber
           = Enroll.classNumber
ORDER BY   stuId ASC;
```

  Result:

stuId	grade
S1002	B
S1010	
S1020	A

- Example 9. Natural Join of Three Tables

  Question: Find course numbers and the names and majors of all students enrolled in the courses taught by `Faculty` member F110.

  Solution: As in the previous example, we need to start at the `Class` table to find the `classNumber` of all courses taught by F110. We then compare these with `classNumber` values in the `Enroll` table to find the `stuId` values of all students in those courses. Then we look at the `Student` table to find the names and majors of all the students enrolled in them.

SQL Query

```
SELECT  Enroll.classNumber, lastName,
        firstName, major
FROM    Class, Enroll, Student
WHERE   facId = 'F110'
        AND Class.classNumber =
        Enroll.classNumber
        AND Enroll.stuId = Student.stuId;
```

Result:

classNumber	lastName	firstName	major
MTH101B	Rivera	Jane	CSC
MTH103C	Burns	Edward	Art
MTH103C	Chin	Ann	Math

This was a natural join of three tables, and it required two sets of common columns. We used the condition of equality for both of the sets in the WHERE line. You may have noticed that the order of the table names in the FROM line corresponded to the order in which they appeared in our plan of solution, but that is not necessary. SQL ignores the order in which the tables are named in the FROM line. The same is true of the order in which we write the various conditions that make up the predicate in the WHERE line. Most sophisticated relational database management systems choose which table to use first and which condition to check first, using an optimizer to identify the most efficient method of accomplishing any retrieval before choosing a plan.

- Example 10. Use of Aliases

  Question: Get a list of all courses that meet in the same room, with their schedules and room numbers.

  Solution: This requires comparing the `Class` table with itself, and it would be useful if there were two copies of the table so we could do a natural join. We can pretend that there are two copies of a table by giving it two "aliases," for example, COPY and COPY2, and then treating these names as if they were the names of two distinct tables. We introduce the "aliases" in the FROM line by writing them immediately after the real table names. Then we have the aliases available for use in the other lines of the query.

SQL Query:

```
SELECT COPYl.classNumber, COPYl.schedule, COPYl.room,
COPY2.classNumber, COPY2.schedule
FROM    Class COPYl, Class COPY2
WHERE   COPYl.room = COPY2.room
        AND COPYl.classNumber > COPY2.classNumber ;
```

Result:

COPYl.classNumber	COPYl.schedule	COPYl.room	COPY2.classNumber	COPY2.schedule
ART103A	MWF9	H221	HST205A	MWF11
CSC201A	TUTHF10	M110	CSC203A	MTHF12
MTH101B	MTUTH9	H225	MTH103C	MWF11

Notice we had to use the qualified names in the SELECT line even before we introduced the "aliases." This is necessary because every column in the `Class` table now appears twice, once in each copy. We added the second condition "COPYl.`classNumber` < COPY2.C0URSE#" to keep every course from being included, since every course obviously satisfies the requirement that it meets in the same room as itself. It also keeps records with the two courses reversed from appearing. For example, because we have,

ART103A MWF9 H221 HST205A MWF11

we do not need the record

HST205A MWF11 H221 ART103A MWF9

Incidentally, we can introduce aliases in any SELECT, even when they are not required.

- Example 11. Join without Equality Condition

  Question: Find all combinations of students and `Faculty` where the student's major is different from the `Faculty` member's department.

  Solution: This unusual request is to illustrate a join in which the condition is not an equality on a common field. In this case, the fields we are examining, `major` and `department`, do not even have the same name. However, we can compare them since they have the same domain. Since we are not told which columns to show in the result, we use our judgment.

SQL Query:

```
SELECT   stuId, lastName, firstName, major, facId,
         name, department
FROM     Student, Faculty
WHERE    Student.major <> Faculty.department;
```

Result:

```
stuId  lastName  firstName major    facId name    department
-----------------------------------------------------------
S1001  Smith     Tom       History  F101  Adams   Art
S1001  Smith     Tom       History  F105  Tanaka  CS
S1001  Smith     Tom       History  F110  Byrne   Math
S1001  Smith     Tom       History  F221  Smith   CS
S1010  Burns     Edward    Art      F202  Smith   History
..........................................
..........................................
..........................................
S1013  McCarthy  Owen      Math     F221  Smith   CS
```

As in relational algebra, a join can be done on any two tables by simply forming the Cartesian product. Although we usually want the natural join as in our previous examples, we might use any type of predicate as the condition for the join. If we want to compare two columns, however, they must have the same domains. Notice that we used qualified names in the WHERE line. This was not really necessary, because each column name was unique, but we did so to make the condition easier to follow.

- Example 12. Using a Subquery with Equality

  Question: Find the numbers of all the courses taught by Byrne of the math department.

  Solution: We already know how to do this by using a natural join, but there is another way of finding the solution. Instead of imagining a join from which we choose records with the same `facId`, we could visualize this as two separate queries. For the first one, we would go to the `Faculty` table and find the record with name of Byrne and `department` of Math. We could make a note of the corresponding `facId`. Then we could take the result of that query, namely Fll0, and search the `Class` table for records with that value in `facId`. Once we found them, we would display the `classNumber`. SQL allows us to sequence these queries so that the result of the first can be used in the second, shown as follows:

SQL Query:

```
SELECT    classNumber
FROM      Class
WHERE     facId =
          (SELECT   facId
          FROM      Faculty
          WHERE     name  = 'Byrne'
                    AND department = 'Math');
```

Result:

```
classNumber
MTH101B
MTH103C
```

Note that this result could have been produced by the following SQL query, using a join:

```
SELECT    classNumber
FROM      Class, Faculty
WHERE     name = 'Byrne' AND department = 'Math'
          AND Class.facId = Faculty.facId;
```

A subquery can be used in place of a join, provided the result to be displayed is contained in a single table and the data retrieved from the subquery consists of only one column. When you write a subquery involving two tables, you name only one table in each SELECT. The query to be done first, the subquery, is the one in parentheses, following the first WHERE line. The main query is performed using the result of the subquery. Normally you want the value of some field in the table mentioned in the main query to match the value of some field from the table in the subquery. In this example, we knew we would get only one value from the subquery, since `facId` is the key of `Faculty`, so a unique value would be produced. Therefore, we were able to use equality as the operator. However, conditions other than equality can be used. Any single comparison operator can be used in a subquery from which you know a single value will be produced. Since the subquery is performed first, the SELECT . . . FROM . . . WHERE of the subquery is actually replaced by the value retrieved, so the main query is changed to the following:

```
SELECT    classNumber
FROM      Class
WHERE     facId  = ('F110');
```

- Example 13. Subquery Using ‘IN’

  Question: Find the names and IDs of all `Faculty` members who teach a class in Room H221.

  Solution: We need two tables, `Class` and `Faculty`, to answer this question. We also see that the names and IDs both appear on the `Faculty` table, so we have a choice of a join or a subquery. If we use a subquery, we begin with the `Class` table to find `facId` values for any courses that meet in Room H221. We find two such entries, so we make a note of those values. Then we go to the `Faculty` table and compare the `facId` value of each record on that table with the two ID values from `Class`, and display the corresponding `facId` and `name`.

  SQL Query:

```
SELECT    name, facId
FROM      Faculty
WHERE     facId IN
          (SELECT   facId
          FROM      Class
          WHERE     room = ‘H221’);
```

  Result:

```
name facId
Adams F10l
Smith F202
```

In the WHERE line of the main query we used IN, rather than =, because the result of the subquery is a set of values rather than a single value. We are saying we want the `facId` in `Faculty` to match any member of the set of values we obtain from the subquery. When the subquery is replaced by the values retrieved, the main query becomes:

```
SELECT    name, facId
FROM      Faculty
WHERE     FAClD IN (‘F101’,‘F202’);
```

The IN is a more general form of subquery than the comparison operator, which is restricted to the case where a single value is produced. We can also use the negative form ‘NOT IN’, which will evaluate to true if the record has a field value which is not in the set of values retrieved by the subquery.

- Example 14. Nested Subqueries

  Question: Get an alphabetical list of names and IDs of all students in any class taught by F110.

  Solution: We need three tables, `Student`, `Enroll`, and `Class`, to answer this question. However, the values to be displayed appear on one table, `Student`, so we can use a subquery. First we check the `Class` table to find the `classNumber` of all courses taught by F110. We find two values, MTH101B and MTH103C. Next we go to the `Enroll` table to find the `stuId` of all students in either of these courses. We find three values, S1020, S1010, and S1002. We now look at the `Student` table to find the records with matching `stuId` values, and display the `stuId`, `lastName`, and `firstName`, in alphabetical order by name.

  SQL Query:

```
SELECT    lastName, firstName, stuId
FROM      Student
WHERE     stuId IN
          (SELECT   stuId
          FROM      Enroll
          WHERE     classNumber IN
                    (SELECT   classNumber
                    FROM      Class
                    WHERE     facId = 'F110'))
ORDER BY lastName, firstName ASC;
```

Result:

lastName	firstName	stuId
Burns	Edward	Sl0l0
Chin	Ann	S1002
Rivera	Jane	S1020

In execution, the most deeply nested SELECT is done first, and it is replaced by the values retrieved, so we have:

```
SELECT    lastName, firstName, stuId
FROM      Student
WHERE     stuId IN
          (SELECT   stuId
          FROM      Enroll
```

```
                WHERE    classNumber IN
                         ('MTH10lB', 'MTH103C'))
ORDER BY lastName, firstName ASC;
```

Next the subquery on `Enroll` is done, and we get:

```
SELECT    lastName, firstName, stuId
FROM      Student
WHERE     stuId IN
          ('S1020', 'Sl0l0', 'S1002')
ORDER BY  lastName, firstName ASC;
```

Finally, the main query is done, and we get the result shown earlier. Note that the ordering refers to the final result, not to any intermediate steps. Also note that we could have performed either part of the operation as a natural join and the other part as a subquery, mixing both methods.

- Example 15. Query Using EXISTS

  Question: Find the names of all students enrolled in CSC201A.

  Solution: We already know how to write this using a join or a subquery with IN. However, another way of expressing this query is to use the existential quantifier, EXISTS, with a subquery.

  SQL Query:

```
SELECT    lastName, firstName
FROM      Student
WHERE     EXISTS
          (SELECT   *
          FROM      Enroll
          WHERE     Enroll.stuId = Student.stuId
          AND       classNumber = 'CSC201A');
```

  Result:

lastName	firstName
Rivera	Jane
Chin	Ann

This query could be phrased as "Find the `lastName` and `firstName` of all students such that there exists an `Enroll` record containing their `stuId` with a `classNumber` of CSC201A". The test for inclusion is the existence of such a record. If it exists, the "EXISTS (SELECT FROM . . .;" evaluates to true.

Notice we needed to use the name of the main query table (`Student`) in the subquery to express the condition `Student.stuId = Enroll.stuId`. In general, we avoid mentioning a table not listed in the FROM for that particular query, but it is necessary and permissible to do so in this case. This form is called a **correlated** subquery, since the table in the subquery is being compared to the table in the main query.

- Example 16. Query Using NOT EXISTS

  Question: Find the names of all students who are not enrolled in CSC201A.

  Solution: Unlike the previous example, we cannot readily express this using a join or an IN subquery. Instead, we will use NOT EXISTS.

  SQL Query:

```
SELECT    lastName, firstName
FROM      Student
WHERE     NOT EXISTS
          (SELECT
          FROM     Enroll
          WHERE    Student.stuId = Enroll.stuId
          AND      classNumber = 'CSC201A');
```

  Result:

lastName	firstName
Smith	Tom
Burns	Edward
Jones	Mary
McCarthy	Owen

We could phrase this query as “Select student names from the `Student` table such that there is no `Enroll` record containing their STUlD values with `classNumber` of CSC201A.”

6.4.3 SELECT with Other Operators

- Example 17. Query Using UNION

  Question: Get IDs of all `Faculty` who are assigned to the history department or who teach in Room H221.

Solution: It is easy to write a query for either of the conditions, and we can combine the results from the two queries by using a UNION operator. The UNION in SQL is the standard relational algebra operator for set union, and works in the expected way, eliminating duplicates.

SQL Query:

```
SELECT    facId
FROM      Faculty
WHERE     department = 'History'
UNION
SELECT    facId
FROM      Class
WHERE     room ='H221';
```

Result:

```
facId
F115
F101
```

- Example 18. Using Functions

  Question: Find the total number of students enrolled in ART103A.

  Solution: Although this is a simple question, we are unable to express it as an SQL query at the moment, because we have not yet seen any way to operate on collections of rows or columns. We need some functions to do so. SQL has five built-in functions: COUNT, SUM, AVG, MAX, and MIN. We will use COUNT, which returns the number of values in a column.

  SQL Query:

```
SELECT    COUNT (DISTINCT stuId)
FROM      Enroll
WHERE     classNumber = 'ART103A';
```

  Result:

  3

The built-in functions operate on a single column of a table. Each of them eliminates null values first, and operates only on the remaining non-null values. The functions return a single value, defined as follows:

COUNT	returns the number of values in the column
SUM	returns the sum of the values in the column

AVG	returns the mean of the values in the column
MAX	returns the largest value in the column
MIN	returns the smallest value in the column.

COUNT, MAX, and MIN apply to both numeric and nonnumeric fields, but SUM and AVG can be used on numeric fields only. The collating sequence is used to determine order of nonnumeric data. If we want to eliminate duplicate values before starting, we use the word DISTINCT before the column name in the SELECT line. COUNT(*) is a special use of the COUNT. Its purpose is to count all the rows of a table, regardless of whether null values or duplicate values occur. Except for COUNT(*). we must always use DISTINCT with the COUNT function, as we did in the above example. If we use DISTINCT with MAX or MIN it will have no effect, because the largest or smallest value remains the same even if two tuples share it. However, DISTINCT usually has an effect on the result of SUM or AVG, so the user should understand whether or not duplicates should be included in computing these. Function references appear in the SELECT line of a query or a subquery.

Additional Function Examples:

Example (a) Find the number of departments that have `Faculty` in them. Because we do not wish to count a department more than once, we use DISTINCT here.

```
SELECT   COUNT(DISTINCT department)
FROM     Faculty;
```

Example (b) Find the average number of credits students have. We do not want to use DISTINCT here, because if two students have the same number of credits, both should be counted in the average.

```
SELECT   AVG(credits)
FROM     Student;
```

Example (c) Find the student with the largest number of credits. Because we want the student's credits to equal the maximum, we need to find that maximum first, so we use a subquery to find it.

```
SELECT   stuId, lastName, firstName
FROM     Student
WHERE    credits =
         (SELECT   MAX(credits)
         FROM      Student);
```

Example (d) Find the ID of the student(s) with the highest grade in any course. Because we want the highest grade, it might appear that we should use the MAX function here. A closer look at the table reveals that the grades are letters A, B, C, etc. For this scale, the best grade is the one that is earliest in the alphabet, so we actually want MIN. If the grades were numeric, we would have wanted MAX.

```
SELECT   stuId
FROM     Enroll
WHERE    grade =
         (SELECT   MIN(grade)
         FROM      Enroll);
```

Example (e) Find names and IDs of students who have less than the average number of credits.

```
SELECT   lastName, firstName, stuId
FROM     Student
WHERE    credits <
         (SELECT   AVG(credits)
         FROM      Student);
```

- Example 19. Using an Expression and a String Constant

  Question: Assuming each course is three credits list, for each student, the number of courses he or she has completed.

  Solution: We can calculate the number of courses by dividing the number of credits by three. We can use the expression `credits/3` in the SELECT to display the number of courses. Since we have no such column name, we will use a string constant as a label. String constants that appear in the SELECT line are simply printed in the result.

  SQL Query:

```
SELECT   stuId, 'Number of courses =', credits/3
FROM     Student;
```

  Result:

```
stuId
S1001 Number of courses =      30
S1010 Number of courses =      21
S1015 Number of courses =      14
```

```
S1002 Number of courses =   12
S1020 Number of courses =    5
S1013 Number of courses =    3
```

By combining constants, column names, arithmetic operators, built-in functions, and parentheses, the user can customize retrievals.

- Example 20. Use of GROUP BY

  Question: For each course, show the number of students enrolled.

  Solution: We want to use the COUNT function, but need to apply it to each course individually. The GROUP BY allows us to put together all the records with a single value in the specified field. Then we can apply any function to any field in each group, provided the result is a single value for the group.

  SQL Query:

```
SELECT      classNumber, COUNT(*)
FROM        Enroll
GROUP BY    classNumber;
```

Result:

```
classNumber
ART103A     3
CSC201A     2
MTH101B     1
HST205A     1
MTH103C     2
```

Note that we could have used COUNT(DISTINCT `stuId`) in place of COUNT(*) in this query.

- Example 21. Use of HAVING

  Problem: Find all courses in which fewer than three students are enrolled.

  Solution: This is a question about a characteristic of the groups formed in the previous example. HAVING is used to determine which groups have some quality, just as WHERE is used with tuples to determine which records have some quality. You are not permitted to use HAVING without a GROUP BY, and the predicate in the HAVING line must have a single value for each group.

SQL Query:

```
SELECT      classNumber
FROM        Enroll
GROUP BY    classNumber
HAVING      COUNT(*) < 3 ;
```

Result:

classNumber
CSC201A
MTH101B
HST205A
MTH103C

- Example 22. Use of LIKE

  Problem: Get details of all MTH courses.

  Solution: We do not wish to specify the exact course numbers, but we want the first three letters of `classNumber` to be MTH. SQL allows us to use LIKE in the predicate to show a pattern string. for character fields. Records whose specified columns match the pattern will be retrieved.

  SQL Query:

```
SELECT      *
FROM        Class
WHERE       classNumber LIKE 'MTH%';
```

  Result:

classNumber	facId	schedule	room
MTH101B	F110	MTUTH9	H225
MTH103C	F110	MWF11	H225

In the pattern string, we can use the following symbols:

% The percent character stands for any sequence of characters of any length >= 0.
_ The underscore character stands for any single character.

All other characters in the pattern stand for themselves.

Examples:

- **`classNumber` LIKE 'MTH%'** means the first three letters must be MTH, but the rest of the string can be any characters.
- **`stuId` LIKE 'S____ '** means there must be five characters, the first of which must be an S
- **`schedule` LIKE '%9'** means any sequence of characters, of length at least one, with the last character a nine.
- **`classNumber` LIKE '%101%'** means a sequence of characters of any length containing l0l. Note the 101 could be the first, last, or only characters, as well as being somewhere in the middle of the string.
- **`name` NOT LIKE 'A%'** means the name cannot begin with an A.

- Example 23. Use of NULL

  Question: Find the `stuId` and `classNumber` of all students whose grades in that course are missing.

  Solution: We can see from the `Enroll` table that there are two such records. You might think they could be accessed by specifying that the grades are not A, B, C, D, or F, but that is not the case. A null grade is considered to have "unknown" as a value, so it is impossible to judge whether it is equal to or not equal to another grade. If we put the condition "WHERE grade <>'A' AND grade <>'B' AND grade <>'C' AND grade <>'D' AND grade <>'F' " we would get an empty table back, instead of the two records we want. SQL uses the logical expression,

  columnname IS [NOT] NULL

  to test for null values in a column.

  SQL Query:

```
SELECT   classNumber,stuId
FROM     Enroll
WHERE    grade IS NULL;
```

  Result:

classNumber	stuId
ART103A	S1010
MTH103C	S1010

Notice that it is illegal to write "WHERE `grade` = NULL," because a predicate involving comparison operators with NULL will evaluate to "unknown" rather than "true" or "false." Also, the WHERE line is the only one on which NULL can appear in a SELECT statement.

- Example 24. Recursive Queries

  SQL:1999 allows recursive queries, which are queries that execute repeatedly until no new results are found. For example, consider a CSCCOURSE table, as shown in Figure 6.4(a). Its structure is:

  CSCCourse(<u>courseNumber</u>, courseTitle, credits, *prerequisiteCourseNumber*)

  For simplicity, we assume a course can have at most one immediate prerequisite course. The prerequisite course number functions as a foreign key for the CSCCourse table, referring to the primary key (course number) of a different course.

  Problem: Find all of a course's prerequisites, including prerequisites of prerequisites for that course.

  SQL Query:

```
WITH RECURSIVE
Prereqs (courseNumber, prerequisiteCourseNumber) AS
   (  SELECT courseNumber, prerequisiteCourseNumber
      FROM CSCCourse
      UNION
      SELECT (COPY1.courseNumber, COPY2.prerequisiteCourseNumber
      FROM Prereqs COPY1, CSCCourse COPY2
      WHERE COPY1.prerequisiteCourseNumber = COPY2.courseNumber);

SELECT *
FROM Prereqs
ORDER BY courseNumber, prerequisiteCourseNumber;
```

This query will display each course number, along with all of that course's prerequisites, including the prerequisite's prerequisite, and so on, all the way back to the initial course in the sequence of its prerequisites. The result is shown in Figure 6.4(b).

6.4.4 Operators for Updating: UPDATE, INSERT, DELETE

The UPDATE operator is used to change values in records already stored in a table. It is used on one table at a time, and can change zero, one, or many records, depending on the predicate. Its form is:

```
UPDATE   tablename
SET      columnname = expression
         [columnname = expression] . . .
[WHERE   predicate];
```

Note that it is not necessary to specify the current value of the field, although the present value may be used in the expression to determine the new value. The SET statement is actually an assignment statement, and works in the usual way.

- Example 1. Updating a Single Field of One Record

  Operation: Change the major of S1020 to Music.

  SQL Command:

```
UPDATE   Student
SET      major = 'Music'
WHERE    stuId = 'S1020';
```

- Example 2. Updating Several Fields of One Record

  Operation: Change Tanaka's department to MIS and rank to Assistant.

  SQL Command:

```
UPDATE   Faculty
SET      department = 'MIS'
         rank = 'Assistant'
WHERE    name = 'Tanaka';
```

CSCCourse

courseNumber	courseTitle	credits	prerequisiteCourseNumber
101	Intro to Computing	3	
102	Computer Applications	3	101
201	Programming 1	4	101
202	Programming 2	4	201
301	Data Structures & Algorithms	3	202
310	Operating Systems	3	202
320	Database Systems	3	301
410	Advanced Operating Systems	3	310
420	Advanced Database Systems	3	320

FIGURE 6.4(a)
CSCCourse table to demonstrate recursive queries

FIGURE 6.4(b)
Result of recursive query

Prereqs

courseNumber	prerequisiteCourseNumber
101	
102	
102	101
201	
201	101
202	
202	101
202	201
301	
301	101
301	201
301	202
310	
310	101
310	201
310	202
320	
320	101
320	201
320	202
320	301
410	
410	101
410	201
410	202
410	310
420	
420	101
420	201
420	202
420	301
420	320

- Example 3. Updating Using NULL

  Operation: Change the major of S1013 from Math to NULL.
  To insert a null value into a field that already has an actual value, we must use the form:

  SET *columnname* = NULL

  SQL Command:

```
UPDATE    Student
SET       major = NULL
WHERE     stuId = 'S1013';
```

- Example 4. Updating Several Records

  Operation: Change grades of all students in CSC201A to A.

  SQL Command:

```
UPDATE    Enroll
SET       grade = 'A'
WHERE     classNumber = 'CSC201A';
```

- Example 5. Updating All Records.

  Operation: Give all students three extra credits.

  SQL Command:

```
UPDATE    Student
SET       credits = credits + 3;
```

Notice we did not need the WHERE line, because all records were to be updated.

- Example 6. Updating with a Subquery

  Operation: Change the room to B220 for all courses taught by Tanaka.

  SQL Command:

```
UPDATE    Class
SET       room = 'B220'
WHERE     facId =
          (SELECT   facId
          FROM      Faculty
          WHERE     name = 'Tanaka');
```

The INSERT operator is used to put new records into a table. Normally, it is not used to load an entire table, because the database management system usually has a load utility to handle that task. However, the INSERT is useful for adding one or a few records to a table. Its form is:

```
INSERT
INTO     tablename [(colname [,colname]. . .)]
VALUES   (constant [,constant] . . .);
```

- Example 1. Inserting a Single Record, with All Fields Specified

  Operation: Insert a new `Faculty` record with ID of F330, name of Jones, department of CSC and rank of Instructor.

  SQL Command:

```
INSERT
INTO     Faculty (facId, name, department, rank)
VALUES   ('F330', 'Jones', 'CSC', 'Instructor');
```

- Example 2. Inserting a Single Record, without Specifying Fields

  Operation: Insert a new student record with ID of S1030, name of Alice Hunt, major of art, and 12 credits.

  SQL Query:

```
INSERT
INTO     Student
VALUES   ('S1030', 'Hunt', 'Alice', 'Art', 12);
```

Notice it was not necessary to specify field names, because the system assumes we mean all the fields in the table. We could have done the same for the previous example.

- Example 3. Inserting a Record with Null Value in a Field

  Operation: Insert a new student record with ID of S1031, name of Maria Bono, zero credits, and no major.

  SQL Command:

```
INSERT
INTO     Student (lastName, firstName, stuId,
         credits)
VALUES   ('Bono', 'Maria', 'S1031', 0);
```

Notice we rearranged the field names, but there is no confusion because it is understood that the order of values matches the order of fields named in the INTO, regardless of their order in the table. Also notice the zero is an actual value for `credits`, not a null value. `major` will be set to null, since we excluded it from the field list in the INTO line.

- Example 4. Inserting Multiple Records

  Operation: Create and fill a new table that shows each course and the number of students enrolled in it.

  SQL Command:

```
CREATE TABLE Enrollment
   (classNumber CHAR(7) NOT NULL,
   Students SMALLINT);
INSERT
INTO Enrollment        (classNumber, Students)
   SELECT              classNumber, COUNT(*)
   FROM                Enroll
   GROUP BY            classNumber;
```

Here, we created a new table, `Enrollment`, and filled it by taking data from an existing table, `Enroll`. If `Enroll` is as it appears in Figure 6.3, `Enrollment` now looks like this:

Enrollment	classNumber	Students
	ART103A	3
	CSC201A	2
	MTH101B	1
	HST205A	1
	MTH103C	2

The `Enrollment` table is now available for the user to manipulate, just as any other table would be. It can be updated as needed, but it will not be updated automatically when the `Enroll` table is updated.

The DELETE is used to erase records. The number of records deleted may be zero, one, or many, depending on how many satisfy the predicate. The form of this command is:

```
DELETE
FROM          tablename
WHERE         predicate;
```

- Example 1. Deleting a Single Record

  Operation: Erase the record of student S1020.

  SQL Command:

  ```
  DELETE
  FROM     Student
  WHERE    stuId = 'S1020';
  ```

- Example 2. Deleting Several Records

  Operation: Erase all enrollment records for student S1020.

  SQL Command:

  ```
  DELETE
  FROM     Enroll
  WHERE    stuId = 'S1020';
  ```

- Example 3. Deleting All Records from a Table

  Operation: Erase all the class records.

If we delete class records and allow their corresponding Enroll records to remain, we would lose referential integrity, because the `Enroll` records would then refer to classes that no longer exist. However, if we have created the tables using Oracle and the commands shown in Figure 6.2, the delete command will not work on the Class table unless we first delete the Enroll records for any students registered in the class, because we wrote,

```
CONSTRAINT Enroll_classNumber_fk FOREIGN KEY (classNumber)
REFERENCES Class (classNumber)
```

for the Enroll table. However, assuming that we have deleted the Enroll records, then we can delete `Class` records.

  SQL Command:

  ```
  DELETE
  FROM     Class;
  ```

This would remove all records from the `Class` table, but its structure would remain, so we could add new records to it at any time.

- Example 4. DELETE with a Subquery

  Operation: Erase all enrollment records for Owen McCarthy.

SQL Command:

```
DELETE
FROM      Enroll
WHERE     stuId =
          (SELECT    stuId
          FROM       Student
          WHERE      lastName = 'Mc Carthy'
                     AND firstName = 'Owen');
```

Because there were no such records, this statement will have no effect on `Enroll`.

6.5 Active Databases

The DBMS has more powerful means of insuring integrity in a database then the column and table constraints discussed in Section 6.3. An **active database** is one in which the DBMS monitors the contents in order to prevent illegal states from occurring, using constraints and triggers.

6.5.1 Enabling and Disabling Constraints

The column and table constraints described in Section 6.3 are identified when the table is created, and are checked whenever a change is made to the database, to ensure that the new state is a valid one. Changes that could result in invalid states include insertion, deletion, and updating of records. Therefore, the constraints are checked by the DBMS whenever one of these operations is performed, usually after each SQL INSERT, DELETE, or UPDATE statement. This is called the IMMEDIATE mode of constraint checking, and it is the default mode. However, there are times when a transaction or application involves several changes and the database will be temporarily invalid while the changes are in progress—some but not all of the changes already have been made. For example, assume we have a Department table in our University example with a `chairPerson` column in which we listed the `facId` of the chairperson, and we use a NOT NULL specification for that field, and make it a foreign key, by writing:

```
CREATE TABLE Department (
   deptName                  CHAR(20),
   chairPerson       CHAR(20) NOT NULL,
   CONSTRAINT Department_deptName_pk PRIMARY KEY(deptName),
   CONSTRAINT Department_facId_fk FOREIGN KEY REFERENCES Faculty (facId));
```

For the Faculty table definition in Figure 6.2, we already have a NOT NULL for the department, but now we could make `department` a foreign key with the new `Department` table as the home table, by writing:

```
ALTER TABLE Faculty ADD CONSTRAINT Faculty_department_fk FOREIGN KEY
REFERENCES Department(deptName);
```

What will happen when we try to create a new department and add the faculty for that department to the Faculty table? We cannot insert the department record unless we already have the chairperson's record in the Faculty table, but we cannot insert that record unless the department record is already there, so each SQL INSERT statement would fail. For such situations, SQL allows us to defer the checking until the end of the entire transaction, using statements such as,

SET CONSTRAINT Department_facId_fk DEFERRED;

or,

SET CONSTRAINT Faculty_department_fk DEFERRED;

We can enable or disable constraints to allow such transactions to succeed, using the statement such as:

DISABLE CONSTRAINT Department_facId_fk;

At the end of the transaction, we should write,

ENABLE CONSTRAINT Department_facId_fk;

to allow enforcement again. Although not recommended, Oracle allows us to write,

DISABLE ALL CONSTRAINTS;

which suspends all integrity checking until a corresponding,

ENABLE ALL CONSTRAINTS;

command is encountered. These statements can be used both interactively and within applications.

6.5.2 SQL Triggers

Like constraints, **triggers** allow the DBMS to monitor the database. However, they are more flexible than constraints, apply to a broader range of situations, and allow a greater variety of actions. A trigger consists of three parts:

- An **event,** which is normally some change made to the database

- A **condition,** which is a logical predicate that evaluates to true or false
- An **action,** which is some procedure that is done when the event occurs and the condition evaluates to true, also called firing the trigger

A trigger has access to the inserted, deleted, or updated data that caused it to fire (i.e., to be activated or raised), and the data values can be used in the code both for the condition and for the action. The prefix `:OLD` is used to refer to the values in a tuple just deleted or to the values replaced in an update. The prefix `:NEW` is used to refer to the values in a tuple just inserted or to the new values in an update. Triggers can be fired either before or after the execution of the insert, delete or update operation. We can also specify whether the trigger is fired just once for each triggering statement, or for each row that is changed by the statement. (Recall that, for example, an update statement may change many rows.) In Oracle, you specify the SQL command (INSERT, DELETE, or UPDATE) that is the event; the table involved; the trigger level (ROW or STATEMENT); the timing (BEFORE or AFTER); and the action to be performed, which can be written as one or more SQL commands in PL/SQL. Section 6.7 discusses PL/SQL in more detail. The Oracle trigger syntax has the form:

```
CREATE OR REPLACE TRIGGER trigger_name
    [BEFORE/AFTER] [INSERT/UPDATE/DELETE] ON table_name
    [FOR EACH ROW] [WHEN condition]
    BEGIN
        trigger body
    END;
```

For example, add to the `Class` table two additional attributes, `currentEnroll`, which shows the number of students actually enrolled in each class, and `maxEnroll`, which is the maximum number of students allowed to enroll. The new table, `RevClass`, is shown in Figure 6.5, along with a new version of the Enroll table, `RevEnroll`. Since `currentEnroll` is a derived attribute, dependent on the `RevEnroll` table, its value should be updated when there are relevant changes made to `RevEnroll`. Changes that affect `currentEnroll` are:

1. A student enrolls in a class
2. A student drops a class
3. A student switches from one class to another

In an active database, there should be triggers for each of these changes. For change (1), we should increment the value of `currentEnroll` by one. We can refer to the `classNumber` of the new `RevEnroll` record by using the prefix :NEW. The corresponding trigger is shown in Figure 6.5(b) Note that we did not use the WHEN because we always want to make this change after a new enrollment record is inserted, so no condition was needed. For change (2), we need to decrement the value of `currentEnroll` by one, and we use the prefix :OLD to refer to the `revEnroll` record being deleted. The trigger is shown in Figure 6.5(c). Change (3) could be treated as a drop followed by an enrollment, but instead we will write a trigger for an update to demonstrate an action with two parts, as shown in Figure 6.5(d).

While these triggers are sufficient if there is room in a class, we also need to consider what should happen if the class is already fully enrolled. The value of `maxEnroll` should have been examined before we allowed a student to enroll in the class, so we need to check that value before we do change (1) or change (3). Assume that if a change would cause a class to be overenrolled, we call a procedure called RequestClosedCoursePermission that takes as parameters the student's ID, the class number, the current enrollment, and maximum enrollment. The action should be taken before we make the change. The trigger is shown in Figure 6.5(e).

Triggers are automatically enabled when they are created. They can be disabled by the statement:

```
ALTER TRIGGER <trigger-name> DISABLE;
```

After disabling, they can enabled again by the statement:

```
ALTER TRIGGER <trigger-name> ENABLE;
```

They can be dropped by the statement:

```
DROP TRIGGER <trigger-name>;
```

Oracle provides an INSTEAD OF form that is especially useful when a user tries to update the database through a view. This form specifies an action to be performed instead of the insert, delete, or update that the user requested. It will be discussed in Section 6.8. Triggers can also be used to provide an audit trail for a table, recording all changes, the time they were made, and the identity of the user who made them, an application that will be discussed in Section 9.6.

RevClass

classNumber	facId	schedule	room	currentEnroll	maxEnroll
ART103A	F101	MWF9	H221	3	25
CSC201A	F105	TuThF10	M110	2	20
CSC203A	F105	MThF12	M110	0	20
HST205A	F115	MWF11	H221	1	35
MTH101B	F110	MTuTh9	H225	1	25
MTH103C	F110	MWF11	H225	2	25

RevEnroll

stuId	classNumber	grade
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

FIGURE 6.5(a)
Tables for Triggers

```
CREATE TRIGGER ADDENROLL
AFTER INSERT ON RevEnroll
FOR EACH ROW
    UPDATE RevClass
    SET currentEnroll = currentEnroll +1
    WHERE RevClass.classNumber = :NEW.classNumber;
```

Figure 6.5(b)
Trigger for Student Enrolling in a Class

Figure 6.5(c)
Trigger for Student Dropping a Class

```
CREATE TRIGGER DROPENROLL
AFTER DELETE ON RevEnroll
FOR EACH ROW
   UPDATE RevClass
   SET currentEnroll = currentEnroll –1
   WHERE RevClass.classNumber = OLD.classNumber;
```

Figure 6.5(d)
Trigger for Student Changing Classes

```
CREATE TRIGGER SWITCHENROLL
AFTER UPDATE OF classNumnber ON RevEnroll
FOR EACH ROW
   BEGIN
      UPDATE RevClass
      SET currentEnroll = currentEnroll + 1
      WHERE RevClass.classNumber = :NEW.classNumber;
      UPDATE RevClass
      SET currentEnroll = currentEnroll –1
      WHERE RevClass.classNumber = :OLD.classNumber;
   END;
```

Figure 6.5(e)
Trigger for Checking for Over-enrollment Before Enrolling Student

```
CREATE TRIGGER ENROLL_REQUEST
BEFORE INSERT OR UPDATE OF classNumber ON RevEnroll
FOR EACH ROW
WHEN
   ((SELECT maxEnroll
   FROM RevClass
   WHERE RevClass.classNumber = :NEW.classNumber)
<
   (SELECT currentEnroll + 1
   FROM RevClass
   WHERE RevClass.classNumber = :NEW.classNumber))
RequestClosedCoursePermission(:NEWstuId, :NEW.classNumber, RevClass.currentEnroll,
RevClass.maxEnroll);
```

6.6 Using COMMIT and ROLLBACK Statements

Any changes made to a database using SQL commands are not permanent until the user writes a COMMIT statement. Discussed in Chapter 10, an SQL transaction ends when either a COMMIT statement or a ROLLBACK statement is encountered. The COMMIT makes permanent the changes made since the beginning of the current transaction, which is either the beginning of the session or the time since the last COMMIT or ROLLBACK. The ROLLBACK undoes changes made by the current transaction. It is wise to write COMMIT often to save changes as you work.

6.7 SQL Programming

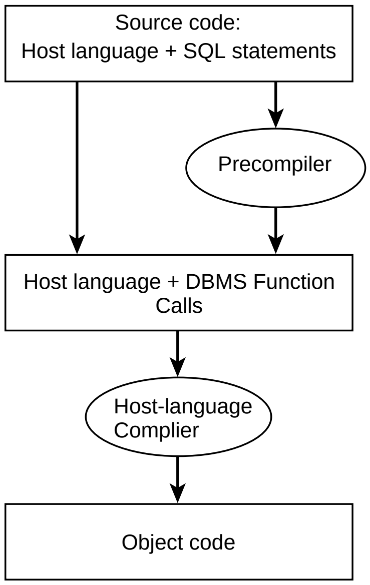
For the interactive SQL statements shown so far, we assumed that there was a user interface that provided an environment to accept, interpret, and execute the SQL commands directly. An example is Oracle's SQLPlus facility. Though some users can interact with the database this way, most database access is through programs.

6.7.1 Embedded SQL

One way that SQL can be used is to embed it in programs written in a general-purpose programming language such as C, C++, Java, COBOL, Ada, Fortran, Pascal, PL/1, or M, referred to as the **host language.** Any interactive SQL command, such as the ones we have discussed, can be used in an application program, with minor modifications. The programmer writes the source code using both host language statements, which provide the control structures, and SQL statements, which manage the database access. Executable SQL statements are preceded by a prefix such as the keyword EXEC SQL, and end with a terminator such as a semicolon. An executable SQL statement can appear wherever an executable host language statement can appear. The DBMS provides a precompiler, which scans the entire program and strips out the SQL statements, identified by the prefix. The SQL is compiled separately into some type of access module, and the SQL statements are replaced, usually by simple function calls in the host language. The resulting host language program can then be compiled as usual. Figure 6.6 illustrates this process.

The data exchange between the application program and the database is accomplished through the host-language program variables, for which

FIGURE 6.6
Processing Embedded SQL Programs


Source code:
Host language + SQL statements
Precompiler
Host language + DBMS Function Calls
Host-language Complier
Object code

attributes of database records provide values or from which they receive their values. These **shared variables** are declared within the program in an SQL declaration section such as the following:

```
EXEC SQL BEGIN DECLARE SECTION;
char stuNumber[5];
char stuLastName[15];
char stuFirstName[12];
char stuMajor[10];
int stuCredits;
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

The first five program variables declared here are designed to match the attributes of the Student table. They can be used to receive values from Student tuples, or to supply values to tuples. Because the data types of the host language might not match those of the database attributes exactly, a type cast operation can be used to pass appropriate values between program variables and database attributes. SQL:1999 provides language bindings that specify the correspondence between SQL data types and the data types of each host language.

The last variable declared, SQLSTATE, is a character array that is used for communicating error conditions to the program. When an SQL library function is called, a value is placed in SQLSTATE indicating which error condition may have occurred when the database was accessed. A value of ‘00000’ indicates no error, while a value ‘02000’ indicates that the SQL statement executed correctly but no tuple was found for a query. The value of SQLSTATE can be tested in host language control statements. For example, some processing may be performed if the value is ‘00000’, or a loop may be written to read a record until the value of ‘02000’ is returned.

Embedded SQL SELECT statements that operate on a single row of a table are very similar to the interactive statements we saw earlier. For example, a SELECT statement that retrieves a single tuple is modified for the embedded case by identifying the shared variables in an INTO line. Note that when shared variables are referred to within an SQL statement, they are preceded by colons to distinguish them from attributes, which might or might not have the same names. The following example begins with a host-language statement assigning a value to the host variable stuNumber. It then uses an SQL SELECT statement to retrieve a tuple from the Student table whose stuId matches the shared variable stuNumber (referred to as :stuNumber in the SQL statement). It puts the tuple’s attribute values into four shared variables that were declared previously:

```
stuNumber = 'S1001';
EXEC SQL  SELECT  Student.lastName, Student.firstName, Student.major,
Student.credits
   INTO :stuLastName, :stuFirstName, :stuMajor, :stuCredits
   FROM Student
   WHERE Student.stuId = :stuNumber;
```

This segment should be followed by a host-language check of the value of SQLSTATE.

To insert a database tuple, we can assign values to shared variables in the host language and then use an SQL INSERT statement. The attribute values are taken from the host variables. For example, we could write:

```
stuNumber = 'S1050';
stuLastName = 'Lee';
stuFirstName = 'Daphne';
stuMajor = 'English';
stuCredits = 0;
EXEC SQL     INSERT
          INTO Student (stuId, lastName, firstName, major, credits)
          VALUES(:stuNumber,:stuLastName, :stuFirstName,:stuMajor,
:stuCredits);
```

We can delete any number of tuples that we can identify using the value of a host variable:

```
stuNumber = 'S1015';
EXEC SQL     DELETE
         FROM Student
         WHERE stuId = :stuNumber;
```

We can also update any number of tuples in a similar manner:

```
stuMajor = 'History';
[FOR {READ ONLY | UPDATE OF attributeNames}];
```

For example, to create a cursor that will later be used to go through CSC student records that we plan to retrieve only, we would write:

```
EXEC SQL DECLARE CSCstuCursor CURSOR FOR
  SELECT stuId, lastName, firstName, major, credits
  FROM student
  WHERE major='CSC';
```

Note that this is a declaration, not an executable statement. The SQL query is not executed yet. After declaring the cursor, we write a statement to open it. This executes the query so that the results multi-set is created. Opening the cursor also positions it just before the first tuple of the results set. For this example, we write:

```
EXEC SQL OPEN CSCstuCursor;
```

To retrieve the first row of the results, we then use the FETCH command which has the form,

```
EXEC SQL FETCH cursorname INTO hostvariables;
```

as in,

```
EXEC SQL FETCH CSCstuCursor INTO :stuNumber,:stuLastName,
:stuFirstName,:stuMajor,:stuCredits
```

The FETCH statement advances the cursor and assigns the values of the attributes named in the SELECT statement to the corresponding shared variables named in the INTO line. A loop controlled by the value of SQL-STATE (e.g., WHILE (SQLSTATE = 00000)) should be created in the host language so that additional rows are accessed. The loop also contains host language statements to do whatever processing is needed. After all data has been retrieved, we exit the loop, and close the cursor in a statement such as:

```
EXEC SQL CLOSE CSCstuCursor;
```

If we plan to update multiple rows using the cursor, we must initially declare it to be updatable, by using a more complete declaration such as:

```
EXEC SQL DECLARE stuCreditsCursor CURSOR FOR
     SELECT stuId, credits
     FROM Student
FOR UPDATE OF credits;
```

We must name in both the SELECT statement and the update attribute list any attribute that we plan to update using the cursor. Once the cursor is open and active, we can update the tuple at which the cursor is positioned, called the **current** of the cursor, by writing a command such as:

```
EXEC SQL UPDATE Student
SET credits = credits +3
WHERE CURRENT OF stuCreditsCursor;
```

Similarly, the current tuple can be deleted by a statement such as:

```
EXEC SQL DELETE FROM Student
WHERE CURRENT OF stuCreditCursor;
```

Users may wish to be able to specify the type of access they need at run time rather than in a static fashion using compiled code of the type just described. For example, we might want a graphical front end where the user could enter a query that can be used to generate SQL statements that are executed dynamically, like the interactive SQL described earlier. Besides the version of SQL just discussed, which is classified as static, there is a **dynamic SQL,** which allows the type of database access to be specified at run time rather than at compile time. For example, the user may be prompted to enter an SQL command that is then stored as a host-language string. The SQL PREPARE command tells the database management system to parse and compile the string as an SQL command, and to assign the resulting executable code to a named SQL variable. An EXECUTE command is then used to run the code. For example, in the following segment the host-language variable `userString` is assigned an SQL update command, the corresponding code is prepared and bound to the SQL identifier `userCommand`, and then the code is executed.

```
char userString[]='UPDATE Student SET credits = 36 WHERE stuId= S1050';
EXEC SQL PREPARE userCommand FROM :userString;
EXEC SQL EXECUTE userCommand;
```

6.7.2 API, ODBC, and JDBC

For embedded SQL, a precompiler supplied by the DBMS compiles the SQL code, replacing it with function calls in the host language. A more flexible approach is for the DBMS to provide an **Application Programming Interface,** API, that includes a set of standard library functions for database processing. The library functions can then be called by the application, which is written in a general purpose language. Programmers use these library functions in essentially the same way they use the library of the language itself. The functions allow the application to perform standard operations such as connecting to the database, executing SQL commands, presenting tuples one at a time, and so on. However, the application would still have to use the precompiler for a particular DBMS and be linked to the API library for that DBMS, so the same program could not be used with another DBMS.

Open Database Connectivity (**ODBC**) and **Java Database Connectivity** (**JDBC**) provide standard ways of integrating SQL code and general purpose languages by providing a common interface. This standardization allows applications to access multiple databases using different DBMSs. The standard provides a high degree of flexibility, allowing development of client-server applications that work with a variety of DBMSs, instead of being limited to a particular vendor API. Most vendors provide ODBC or JDBC drivers that conform to the standard. An application using one of these standard interfaces can use the same code to access different databases without recompilation. ODBC/JDBC architecture requires four components—the application, driver manager, driver, and data source (normally a database), as illustrated in Figure 6.7. The application initiates the connection with the database, submits data requests as SQL statements to the DBMS, retrieves the results, performs processing, and terminates the connection, all using the standard API. A driver manager loads and unloads drivers at the application's request, and passes the ODBC or JDBC calls to the selected driver. The database driver links the application to the data source, translates the ODBC or JDBC calls to DBMS-specific calls, and handles data translation needed because of any differences between the DBMS's data language and the ODBC/JDBC standard, and error-handling differences that arise between the data source and the standard. The data source is the database (or other source, such as a spreadsheet) being accessed, along with its environment, consisting of its DBMS and platform. There are different levels of conformance defined for ODBC and

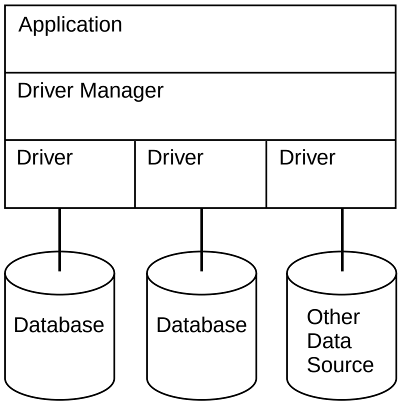


FIGURE 6.7
ODBC/JDBC Architecture

JDBC drivers, depending on the type of relationship between the application and the database.

6.7.3 SQL PSM

Most database management systems include an extension of SQL itself, called **Persistent Stored Modules** (PSM), to allow users to write stored procedures, called **internal routines,** within the database process space, rather than externally. These facilities are used to write SQL routines that can be saved with the database schema and invoked when needed. (In contrast, programs written in a host language are referred to as external routines.) Oracle's PL/SQL, which can be accessed from within the SQLPlus environment, is this type of facility. The SQL/PSM standard is designed to provide complete programming language facilities, including declarations, control structures, and assignment statements.

SQL/PSM modules include functions, procedures, and temporary relations. To declare a procedure, we write:

```
CREATE PROCEDURE procedure_name (parameter_list)
   declarations of local variables
   procedure code
```

Each parameter in the parameter list has three items—mode, name, and datatype. The mode can be IN, OUT, or INOUT, depending on whether it is an input parameter, an output parameter, or both. The name and data type of the parameter must also be given. Next we have declarations of local variables, if any, and the code for the procedure, which may contain

SQL statements of the type we have seen previously for embedded SQL, plus assignment statements, control statements, and error handling.

A function has a similar declaration, as follows:

```
CREATE FUNCTION function_name (parameter list)
  RETURNS SQLdatatype
declarations of local variables
function code (must include a RETURN statement)
```

Functions accept only parameters with mode IN, to prevent side effects. The only value returned should be the one specified in the RETURN statement.

Declarations have this form,

DECLARE *identifier datatype;*

as in,

```
DECLARE    status               VARCHAR2;
DECLARE    number_of_courses    NUMBER;
```

The SQL assignment statement, SET, permits the value of a constant or an expression to be assigned to a variable. For example, we could write:

```
SET status = 'Freshman'; //However, Oracle uses := for assignment
SET number_of_courses = credits/3;
```

Branches have the form:

```
IF (condition) THEN statements;
  ELSEIF (condition) statements;
  . . .
  ELSEIF (condition) statements;
  ELSE statements;
END IF;
```

for example,

```
IF (Student.credits <30) THEN
  SET status = 'Freshman';
  ELSEIF (Student.credits <60) THEN
    SET status = 'Sophomore';
  ELSEIF (Student.credits <90) THEN
    SET status = 'Junior';
  ELSE SET status = 'Senior';
END IF;
```

The CASE statement can be used for selection based on the value of a variable or expression.

```
CASE selector
    WHEN value1      THEN statements;
    WHEN value2      THEN statements;
    . . .
END CASE;
```

Repetition is controlled by LOOP ... ENDLOOP, WHILE ... DO ... END WHILE REPEAT, ... UNTIL ... END REPEAT, and FOR ... DO ... END FOR structures. We can use cursors as we did for embedded SQL. For example, we could write:

```
...
DECLARE CSCstuCursor CURSOR FOR
   SELECT stuId, lastName, firstName, major, credits
   FROM student
   WHERE major= 'CSC';
OPEN CSCstuCursor;
WHILE (SQLCODE = '00000') DO
   FETCH CSCstuCursor INTO stuNumber,stuLastName,stuFirstName,
   stuMajor,stuCredits; statements to process these values
END WHILE;
CLOSE CSCstuCursor;
```

The language also provides predefined exception handlers, and allows the user to create user-defined exceptions as well.

Once a procedure has been created, it can be executed by this command:

EXECUTE *procedure_name(actual_ parameter_ list);*

As with most languages, the actual parameter list consists of the values or variables being passed to the procedure, as opposed to the formal parameter list that appears in the declaration of the procedure.

A function is invoked by using its name, typically in an assignment statement. For example:

```
SET newVal = MyFunction (val1, val2);
```

6.8 Creating and Using Views

Views are an important tool for providing users with a simple, customized environment and for hiding data. As explained in Section 6.2, a relational view does not correspond exactly to the general external view, but is a virtual table derived from one or more underlying base tables. It does not exist in storage in the sense that the base tables do, but is created by selecting specified rows and columns from the base tables, and possibly performing operations on them. The view is dynamically produced as the user works with it. To make up a view, the DBA decides which attributes the user needs to access, determines what base tables contain them, and constructs one or more views to display in table form the values the user should see. Views allow a dynamic external model to be created for the user easily. The reasons for providing views rather than allowing all users to work with base tables are as follows:

- Views allow different users to see the data in different forms, permitting an external model that differs from the conceptual model.
- The view mechanism provides a simple authorization control device, easily created and automatically enforced by the system. View users are unaware of, and cannot access, certain data items.
- Views can free users from complicated DML operations, especially in the case where the views involve joins. The user writes a simple SELECT statement using the view as the named table, and the system takes care of the details of the corresponding more complicated operations on the base tables to support the view.
- If the database is restructured on the conceptual level, the view can be used to keep the user's model constant. For example, if the table is split by projection and the primary key appears in each resulting new table, the original table can always be reconstructed when needed by defining a view that is the join of the new tables.

The following is the most common form of the command used to create a view:

```
CREATE VIEW viewname
     [(viewcolname [,viewcolname] . . .)]
     AS   SELECT    colname [,colname] . . .
          FROM      basetablename [,basetablename] . . .
          WHERE     condition;
```

The view name is chosen using the same rules as the table name, and should be unique within the database. Column names in the view can be different from the corresponding column names in the base tables, but they must obey the same rules of construction. If we choose to make them the same, we need not specify them twice, so we leave out the *viewcolname* line. In the AS SELECT line, we list the names of the columns from the underlying base tables that we wish to include in the view. The order of these names should correspond exactly to the *viewcolnames,* if those are specified. However, columns chosen from the base tables may be rearranged in any desired manner in the view. As in the usual SELECT . . . FROM . . . WHERE, the condition is a logical predicate that expresses some restriction on the records to be included. A more general form of the CREATE VIEW uses any valid subquery in place of the SELECT we have described.

- Example 1. Choosing a Vertical and Horizontal Subset of a Table

  Assume a user needs to see IDs and names of all history majors. We can create a view for this user as follows:

```
CREATE VIEW HISTMAJ (StudentName,StudentId)
 AS   SELECT   lastName, firstName,stuId
      FROM     Student
      WHERE    major = 'History';
```

Here we renamed the columns from our base table. The user of this view need not know the actual column names.

- Example 2. Choosing a Vertical Subset of a Table

If we would like a table of all courses with their schedules and room, we could create this as follows:

```
CREATE VIEW   ClassLoc
AS SELECT     classNumber, schedule, room
FROM          Class;
```

Notice we did not need a condition here, since we wanted these parts of all `Class` records. This time we kept the names of the columns as they appear in the base table.

- Example 3. A View Using Two Tables

Assume a user needs a table containing the IDs and names of all students in course CSC101. The virtual table can be created by choosing records in `Enroll` that have `classNumber` of CSC101, matching the `stuId` of those records with the stuId of the Student records, and taking the corresponding lastName, and firstName from Student. We could express this as a join or a subquery.

```
CREATE VIEW ClassList
  AS    SELECT   Student.stuId,lastName,
                 firstName
        FROM     Enroll,Student
        WHERE    classNumber = 'CSCl0l'
                 AND Enroll.stuId =
                 Student.stuId;
```

- Example 4. A View of a View

We can define a view derived from a view. For example, we can ask for a subset of the ClassLoc (virtual) table by writing:

```
CREATE VIEW ClassLoc2
  AS    SELECT   classNumber, room
        FROM     ClassLoc;
```

- Example 5. A View Using a Function

In the SELECT statement in the AS line we can include built–in functions and GROUP BY options. For example, if we want a view of Enroll that gives classNumber and the number of students enrolled in each class, we write:

```
CREATE VIEW ClassCount (classNumber, TotCount)
  AS    SELECT   classNumber, COUNT(*)
        FROM     Enroll
                 GROUP BY classNumber;
```

Notice we had to supply a name for the second column of the view, since there was none available from the base table.

- Example 6. Operations on Views

Once a view is created, the user can write SELECT statements to retrieve data through the view. The system takes care of mapping the user names to the underlying base table names and column names, and performing whatever functions are required to produce the result in the form the user expects. Users can write SQL queries that refer to joins, ordering, grouping, built-in functions, and so on, of views just as if they were operating on base tables. Since the SELECT operation does not change the underlying base tables, there is no restriction on using it with views. The following is an example of a SELECT operation on the ClassLoc view:

```
SELECT  *
FROM    ClassLoc
WHERE   room LIKE 'H%';
```

INSERT, DELETE, and UPDATE present certain problems with views. For example, suppose we had a view of student records such as:

```
StudentVwl(lastName, firstName, major, credits)
```

If we were permitted to insert records, any records created through this view would actually be `Student` records, but would not contain `stuId`, which is the key of the `Student` table. Since `stuId` would have the NOT NULL constraint, we would have to reject any records without this field. However, if we had the following view,

```
StudentVw2(stuId,lastName, firstName,credits)
```

we should have no problem inserting records, since we would be inserting `Student` records with a null major field, which is allowable. We could accomplish this by writing:

```
INSERT
INTO    StudentVw2
VALUES  ('S1040'. 'Levine', 'Adam', 30);
```

However, the system should actually insert the record into the Student table. We can use an INSTEAD OF trigger to make sure this happens.

```
CREATE TRIGGER InsertStuVw2
   INSTEAD OF INSERT ON StudentVw2
   FOR EACH ROW
        INSERT
        INTO Student
        VALUES (:NEW.stuId, :NEW.lastName,
        :NEW.firstName, NEW. Credits);
```

Now let us consider inserting records into the view `ClassCount`, as described earlier in Example 5. This view used the COUNT function on groups of records in the `Enroll` table. Obviously, this table was meant to be a dynamic summary of the `Enroll` table, rather than being a row and column subset of that table. It would not make sense for us to permit new,

```
ClassCount
```

records to be inserted, since these do not correspond to individual rows or columns of a base table.

The problems we have identified for INSERT apply with minor changes to UPDATE and DELETE as well. As a general rule, these three operations can be performed on views that consist of actual rows and columns of underlying base tables, provided the primary key is included in the view, and no other constraints are violated. INSTEAD OF triggers can be used to ensure that the database updates the underlying tables.

6.9 The System Catalog

The **system catalog** or **system data dictionary** can be thought of as a database of information about databases. It contains, in table form, a summary of the structure of each database as it appears at a given time. Whenever a base table, view, index, constraint, stored module, or other item of a database schema is created, altered, or dropped, the DBMS automatically updates its entries in the catalog. The system also uses the catalog to check authorizations and to store information for access plans for applications. Users can query the data dictionary using ordinary SQL SELECT commands. However, since the data dictionary is maintained by the DBMS itself, the SQL UPDATE, INSERT, and DELETE commands cannot be used on it.

The **Oracle data dictionary** contains information about all schema objects, but access to it is provided through three different views, called USER, ALL, and DBA. In Oracle, each user is automatically given access to all the objects he or she creates. The **USER view** provides a user with information about all the objects created by that user. Users can be given access to objects that are created by others. The **ALL view** provides information about those objects in addition to the one the user has created. The **DBA view,** which provides information about all database objects, is available to the database administrator. Each of the views is invoked by using the appropriate term as a prefix for the object(s) named in the FROM clause in a query. For example, if a user wants a list of the names of all the tables he or she has created, the query is:

```
SELECT TABLE_NAME
FROM USER_TABLES;
```

Queries can be written by using the appropriate prefix (USER_, ALL_, DBA_) in the FROM clause, followed by one of the categories CATALOG, CONSTRAINTS, CONS_COLUMNS (columns that have constraints), DICTIONARY, IND_COLUMNS (columns that have indexes), INDEXES, OBJECTS, TAB_COLUMNS (table columns), TRIGGERS, ROLES, PROFILES, SEQUENCES, SOURCE (source code for a module), SYS_PRIVS, USERS, TABLES, TABLESPACES, VIEWS, and other objects in the schema. For each of these categories, each of the three views (USER, ALL, DBA) has several columns. Generally, you can assume that each category has a column for the name of the object, which you can use to write a query such as:

```
SELECT VIEW_NAME
FROM USER_VIEWS;
```

To determine what all the available columns are, you can use the wild card (*) in the SELECT clause. For example,

```
SELECT*
FROM USER_TAB_COLUMNS;
```

will display all the recorded information about the columns in the tables you have created. Then you can use the view's column names (e.g., COLUMN_NAME, DATA_TYPE) in a more targeted query, such as:

```
SELECT COLUMN_NAME, DATA_TYPE
FROM USER_TAB_COLUMNS
WHERE TABLE_NAME = 'STUDENT';
```

Another way to learn about the objects is to use the DESCRIBE command. Once you know the name of an object (e.g., a table, constraint, column), which you can obtain by one of the methods just illustrated, you can ask for a description of it. For example, you can write,

```
DESCRIBE STUDENT;
```

to see what is known about the Student table, or,

```
DESCRIBE HISTMAJ;
```

to see what is known about the HISTMAJ view that we created in Section 6.8. If we want to learn what information is available about constraints in the USER view, we would write:

```
DESCRIBE USER_CONSTRAINTS;
```

We can then write a query using the names of the columns that are displayed, such as:

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME
FROM USER_CONSTRAINTS;
```

For information about triggers, the command,

```
SELECT TRIGGER_NAME, TRIGGER_EVENT, TRIGGER_TYPE
FROM USER_TRIGGERS;
```

provides useful information about them.

IBM's **DB2 Universal Database** has a system catalog that is also kept in the form of tables in a schema called SYSIBM, usually with restricted access. Two views of the tables, SYSCAT and SYSSTAT, are available for users. The **SYSCAT schema** has many tables, all of which are read-only. Some of the most important ones are the following:

```
TABLES (TABSCHEMA, TABNAME, DEFINER, TYPE, STATUS, COLCOUNT,
KEYCOLUMNS, CHECKCOUNT,..)
COLUMNS (TABSCHEMA, TABNAME, COLNAME, COLNO, TYPENAME, LENGTH,
DEFAULT, NULLS, ...)
INDEXES (INDSCHEMA, INDNAME, DEFINER, TABSCHEMA, TABNAME, COLNAMES,
UNIQUERULE, COLCOUNT, ...)
TRIGGERS (TRIGSCHEMA, TRIGNAME, DEFINER, TABSCHEMA, TABNAME,
TRIGTIME, TRIGEVENT, ...)
VIEWS (VIEWSCHEMA, VIEWNAME, DEFINER, TEXT ...)
```

You can write queries for these tables using SQL, as in,

```
SELECT TABSCHEMA, TABNAME
FROM TABLES
WHERE DEFINER = 'JONES';
```

which gives the names of all the tables created by Jones.

The query,

```
SELECT *
FROM COLUMNS
WHERE TABNAME = 'STUDENT'
GROUP BY COLNAME;
```

gives all the available information about the columns of the Student table.

6.10 Chapter Summary

Oracle, IBM's DB2, MySQL, SQL Server and other relational database management systems use **SQL,** a standard relational DDL and DML. On the conceptual level, each relation is represented by a **base table.** The external level consists of **views,** which are created from subsets, combinations, or other operations on the base tables. A base table can have **indexes,** one of which can be a clustered index, defined on it. Dynamic database definition allows the structure to be changed at any time.

SQL DDL commands **CREATE TABLE** and **CREATE INDEX** are used to create the base tables and their indexes. Several built-in **data types** are available, and users can also define new types. **Constraints** can be specified on the column or table level. The **ALTER TABLE** command allows changes to existing tables, such as adding a new column, dropping a column, changing data types, or changing constraints. The **RENAME TABLE** command allows the user to change a table's name. **DROP TABLE** and **DROP INDEX** remove tables and indexes, along with all the data in them, from the database.

The **DML** commands are **SELECT, UPDATE, INSERT,** and **DELETE.** The SELECT command has several forms, and it performs the equivalent of the relational algebra SELECT, PROJECT, and JOIN operations. Options include **GROUP BY, ORDER BY, GROUP BY ... HAVING, LIKE,** and built-in functions **COUNT, SUM, AVG, MAX,** and **MIN.** The SELECT statement can operate on joins of tables, and can handle **subqueries,** including correlated subqueries. Expressions and set operations are also possible. The UPDATE command may be used to update one or more fields in one or more records. The INSERT command can insert one or more records, possibly with null values for some fields. The DELETE operator erases records, while leaving the table structure intact.

An active database is one where the DBMS actively monitors changes to ensure that only legal instances of the database are created. A combination of constraints and **triggers** may be used to create an active database.

The **COMMIT** statement makes permanent all changes that have been made by the current transaction. The **ROLLBACK** statement undoes all changes that were made by the current transaction. The current transaction begins immediately after the last COMMIT or ROLLBACK, or if neither of these occurred, then at the beginning of the current user session.

SQL is often used in a programming environment rather than interactively. It can be **embedded** in a host programming language and separately compiled by a **precompiler.** It can be used with a standard API through **ODBC** or **JDBC.** It can also be used as a complete language using its own **SQL PSM**s. Oracle's PL/SQL is an example of a complete programming environment for creating SQL PSMs.

The **CREATE VIEW** command is used to define a virtual table, by selecting fields from existing base tables or previously defined views. The SELECT operation can be used on views, but other DML commands are restricted to certain types of views. An **INSTEAD OF trigger** is useful for replacing user's DML commands written on a view with corresponding commands on the base table(s) used for the view. A view definition can be destroyed by a DROP VIEW command.

The **system catalog** or **system data dictionary** is a database containing information about the user's database. It keeps track of the tables, columns, indexes, and views that exist, as well as authorization information and other data. The system automatically updates the catalog when structural changes and other modifications are made.

Exercises

6.1 Write the commands needed to create indexes for the `Student`, `Faculty`, `Class`, and `Enroll` tables in this chapter.

6.2 For each of the join examples (Examples 7–11) in Section 6.4.2, replace the join by a subquery, if possible. If not possible, explain why not.

Directions for exercises 6.3–6.25: For the schema that follows, write the indicated commands in SQL. Figure 6.8 shows the DDL for creating these tables. It shows that `departmentName` is a foreign key in the `Worker` table, that `mgrId` is a foreign key in the `Dept` table, that `projMgrId` is a foreign key in the `Project` table, and that `projNo` and `empId` are foreign keys in the `Assign` table. We assume each department has a manager, and each project has a manager, but these are not necessarily related. (Note: It is recommended that you do Lab Exercise 1 in conjunction with these exercises. If Oracle is not available, you can use another relational DBMS, including the freeware MySQL, which you can download. Depending on the product, you may need to make some changes to the DDL. If you do not plan to do Lab Exercise 1, you can simply write out the commands.)

```
Worker (empId, lastName, firstName, departmentName, birthDate,
hireDate, salary)
Dept (departmentName, mgrId)
Project (projNo, projName, projMgrId, budget, startDate,
expectedDurationWeeks)
Assign (projNo, empId, hoursAssigned, rating)
```

6.3 Get the names of all workers in the accounting department.

6.4 Get an alphabetical list of names of all workers assigned to project 1001.

6.5 Get the name of the employee in the research department who has the lowest salary.

6.6 Get details of the project with the highest budget.

6.7 Get the names and departments of all workers on project 1019.

```
CREATE TABLE Worker (
empId NUMBER (6),
lastName VARCHAR2 (20) NOT NULL,
firstName VARCHAR2 (15) NOT NULL,
departmentNumber   VARCHAR2 (15),
birthDate DATE,
hireDate DATE,
salary NUMBER (8, 2),
CONSTRAINT Worker_empid_pk PRIMARY KEY (empid),
CONSTRAINT Worker_departmentNumber FOREIGN KEY (departmentNumber) REFERENCES
Dept (departmentName) ON UPDATE CASCADE ON DELETE SET NULL);

CREATE TABLE Dept (
departmentName VARCHAR2 (15),
mgrId VARCHAR2 (20),
CONSTRAINT Dept_departmentName_pk PRIMARY KEY (departmentName),
CONSTRAINT Dept_mgrId FOREIGN KEY (mgrId) REFERENCES Worker (empId)) ON UPDATE
CASCADE ON
DELETE SET NULL;

CREATE TABLE Project (
projNo NUMBER (6),
projName VARCHAR2 (20),
projMgrId VARCHAR2 (20),
budget NUMBER (8, 2),
startDate DATE,

expectedDurationWeeks NUMBER (4),
CONSTRAINT Project_projNo_pk PRIMARY KEY (projNo),
CONSTRAINT Project_projMgrId_fk FOREIGN KEY (projMgrid) REFERENCES WORKER (empId)
ON UPDATE
CASCADE ON DELETE SET NULL);
```

(continued)

FIGURE 6.8

DDL and Insert Statements for Worker-Project-Assign Example

```
CREATE TABLE Assign (
projNo NUMBER (6),
empId NUMBER (6),
hoursAssigned NUMBER (3),
rating NUMBER (1),
CONSTRAINT Assign_projNo_empId_pk PRIMARY KEY (projNo, empId),
CONSTRAINT Assign_projNo_fk FOREIGN KEY (projNo) REFERENCES Project (projNo) ON UPDATE
CASCADE, ON DELETE NO ACTION,
CONSTRAINT Assign_empId_fk FOREIGN KEY (empId) REFERENCES Worker (empId) ON
UPDATE
CASCADE, ON DELETE CASCADE);

INSERT INTO Dept VALUES ('Accounting',);
INSERT INTO Dept VALUES ('Research',);

INSERT INTO Worker VALUES(101,'Smith', 'Tom','Accounting', '01-Feb-1960', '06-Jun-1983',50000);
INSERT INTO Worker VALUES(103,'Jones','Mary' ,'Accounting', '15-Jun-1965', '20-Sep-1985',48000);
INSERT INTO Worker VALUES(105,'Burns','Jane', 'Accounting', '21-Sep-1970' ,'12-Jun-1990',39000);
INSERT INTO Worker VALUES(110,'Burns','Michael', 'Research', '05-Apr-1967', '10-Sep-1990',70000);
INSERT INTO Worker VALUES(115,'Chin','Amanda', 'Research', '22-Sep-1965', '19-Jun-1985',60000);

UPDATE Dept SET mgrId = 101 WHERE deptName = 'Accounting';
UPDATE Dept SET mgrId = 101 WHERE deptName = 'Research';

INSERT INTO Project VALUES (1001, 'Jupiter', 101, 300000, '01-Feb-2004', 50);
INSERT INTO Project VALUES (1005, 'Saturn', 101, 400000, '01-Jun-2004', 35);
INSERT INTO Project VALUES (1019, 'Mercury', 110, 350000, '15-Feb-2004', 40);
INSERT INTO Project VALUES (1025, 'Neptune', 110, 600000, '01-Feb-3005', 45);
INSERT INTO Project VALUES (1030, 'Pluto', 110, 380000, '15-Sept-2004', 50);

INSERT INTO Assign VALUES (1001, 101, 30,);
INSERT INTO Assign VALUES (1001, 103, 20, 5);
INSERT INTO Assign VALUES (1005, 103, 20,);
INSERT INTO Assign VALUES (1001, 105, 30,);
INSERT INTO Assign VALUES (1001, 115, 20, 4);
INSERT INTO Assign VALUES (1019, 110, 20, 5);
INSERT INTO Assign VALUES (1019, 115, 10, 4);
INSERT INTO Assign VALUES (1025, 110, 10,);
INSERT INTO Assign VALUES (1030, 110, 10,);
```

FIGURE 6.8—Continued

6.8 Get an alphabetical list of names and corresponding ratings of all workers on any project that is managed by Michael Burns.

6.9 Create a view that has project number and name of each project, along with the IDs and names of all workers assigned to it.

6.10 Using the view created in Exercise 6.9, find the project number and project name of all projects to which employee 1001 is assigned.

6.11 Add a new worker named Jack Smith with ID of 1999 to the research department.

6.12 Change the hours, which employee 110 is assigned to project 1019, from 20 to 10.

6.13 For all projects starting after May 1, 2004, find the project number and the IDs and names of all workers assigned to them.

6.14 For each project, list the project number and how many workers are assigned to it.

6.15 Find the employee names and department manager names of all workers who are not assigned to any project.

6.16 Find the details of any project with the word "urn" anywhere in its name.

6.17 Get a list of project numbers and names and starting dates of all projects that have the same starting date.

6.18 Add a field called `status` to the `Project` table. Sample values for this field are `active, completed, planned, cancelled.` Then write the command to undo this change.

6.19 Get the employee ID and project number of all employees who have no ratings on that project.

6.20 Assuming that `salary` now contains annual salary, find each worker's ID, name, and monthly salary.

6.21 Add a field called `numEmployeesAssigned` to the Project table. Use the UPDATE command to insert values into the field to correspond to the current information in the Assign table. Then write a trigger that will update the field correctly whenever an assignment is made, dropped, or updated. Write the command to make these changes permanent.

6.22 a. Write an Oracle data dictionary query to show the names of all the columns in a table called Customers.

6.22 b. Write a corresponding query for the DB2 UDB SYSCAT tables for this example.

6.23 a. Write an Oracle data dictionary query to find all information about all columns named PROJ#.

6.23 b. Write a corresponding query for the DB2 UDB SYSCAT tables for this example.

6.24. a. Write an Oracle data dictionary query to get a list of names of people who have created tables, along with the number of tables each has created. Assume you have DBA privileges.

6.24 b. Write a corresponding query for the DB2 UDB SYSCAT tables for this example.

6.25 a. Write an Oracle data dictionary query to find the names of tables that have more than two indexes.

6.25 b. Write a corresponding query for the DB2 UDB SYSCAT tables for this example.

Lab Exercises

Lab Exercise 6.1. Exploring the Oracle Database for Worker-Dept-Project-Assign Example (script is on CD)

A script to create an Oracle database for the example used in exercises 6.3–6.25 appears in Figure 6.8 and on the CD that accompanies this book. The script was written using Notepad. Find the script on the CD, copy it into your own directory, and open it with Notepad. Open Oracle's SQLPlus facility. You will switch back and forth between Notepad and SQLPlus, because the editor in SQLPlus is hard to use.

a. In Notepad, highlight and copy the command to create the first table, then switch to SQLPlus and paste that command in the SQLPlus window and execute the command. You should see the message "Table created." If you get an error message instead, go back to the Notepad file and correct the error.

b. Continue to create the remaining tables one at a time in the same manner.

c. Run the INSERT commands to populate the tables. Explain why the two UPDATE statements were used.

d. Using that implementation, execute the Oracle SQL statements for Exercises 6.3–6.25.

Lab Exercise 6.2. Creating and Using a Simple Database in Oracle

a. Write the DDL commands to create the Student, Faculty, Class, and Enroll tables for the University database shown in Figure 6.2.

b. Using the INSERT command, add the records as they appear in Figure 6.3 to your new Oracle database.

c. Write SQL queries for the following questions, and execute them.

i. Find the names of all history majors.

ii. Find the class number, schedule, and room for all classes that Smith of the history department teaches.

iii. Find the names of all students who have fewer than average number of credits.

iv. Find the names of all the teachers that Ann Chin has, along with all her classes and midterm grades from each.

v. For each student, find the number of classes he or she is enrolled in.

SAMPLE PROJECT: CREATING AND MANIPULATING A RELATIONAL DATABASE FOR THE ART GALLERY

In the sample project section at the end of Chapter 5, we created a normalized relational model for The Art Gallery database. Renaming the tables slightly, we concluded that the following model should be implemented:

```
(1) Artist (artistId, firstName, lastName, interviewDate,
interviewerName, areaCode, telephoneNumber, street, zip,
salesLastYear, salesYearToDate, socialSecurityNumber, usualMedium,
usualStyle, usualType)

(2) Zips (zip, city, state)

(3) PotentialCustomer (potentialCustomerId, firstname, lastName,
areaCode, telephoneNumber, street, zip, dateFilledIn,
preferredArtistId, preferredMedium, preferredStyle, preferredType)
```

(4) Artwork (artworkId, *artistId,* workTitle, askingPrice, dateListed, dateReturned, dateShown, status, workMedium, workSize, workStyle, workType, workYearCompleted, *collectorSocialSecurityNumber*)

(5) ShownIn (*artworkId,showTitle*)

(6) Collector (socialSecurityNumber, firstName, lastName, street, *zip,* interviewDate, interviewerName, areaCode, telephonenumber, salesLastYear, salesYearToDate, *collectionArtistId,* collectionMedium, collectionStyle, collectionType, SalesLastYear, SalesYearToDate)

(7) Show (showTitle, *showFeaturedArtistId,* showClosingDate, showTheme, showOpeningDate)

(8) Buyer (buyerId, firstName, lastName, street, zip, areaCode, telephoneNumber, purchasesLastYear, purchasesYearToDate)

(9) Sale (InvoiceNumber, *artworkId,* amountRemittedToOwner, saleDate, salePrice, saleTax, *buyerId, salespersonSocialSecurityNumber*)

(10) Salesperson (socialSecurityNumber, firstName, lastName, street, zip)

- Step 6.1. Update the data dictionary and list of assumptions if needed. For each table, write the table name and write out the names, data types, and sizes of all the data items, identify any constraints, using the conventions of the DBMS you will use for implementation.

No changes were made to the list of assumptions. No changes to the listed data items in the data dictionary are needed. For an Oracle database, the tables will have the structures as follows:

TABLE Zips

Item	Datatype	Size	Constraints	Comments
Zip	CHAR	10	PRIMARY KEY	
city	VARCHAR2	15	NOT NULL	
state	CHAR	2	NOT NULL	

TABLE Artist

Item	Datatype	Size	Constraints	Comments
artistId	NUMBER	6	PRIMARY KEY	
firstName	VARCHAR2	15	NOT NULL; (firstName, lastName) UNIQUE	
lastName	VARCHAR2	20	NOT NULL; (firstName, lastName) UNIQUE	
interviewDate	DATE			
interviewerName	VARCHAR2	35		
areaCode	CHAR	3		
telephoneNumber	CHAR	7		
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY REF Zips	
salesLastYear	NUMBER	8,2		
salesYearToDate	NUMBER	8,2		
socialSecurityNumber	CHAR	9	UNIQUE	
usualMedium	VARCHAR	15		
usualStyle	VARCHAR	15		
usualType	VARCHAR	20		

TABLE Collector

Item	Datatype	Size	Constraints	Comments
socialSecurityNumber	CHAR	9	PRIMARY KEY	
firstName	VARCHAR2	15	NOT NULL	
lastName	VARCHAR2	20	NOT NULL	
interviewDate	DATE			
interviewerName	VARCHAR2	35		
areaCode	CHAR	3		
telephoneNumber	CHAR	7		
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY Ref Zips	
salesLastYear	NUMBER	8,2		
salesYearToDate	NUMBER	8,2		
collectionArtistId	NUMBER	6	FOREIGN KEY REF Artist	
collectionMedium	VARCHAR	15		
collectionStyle	VARCHAR	15		
collectionType	VARCHAR	20		

TABLE PotentialCustomer

Item	Datatype	Size	Constraints	Comments
potentialCustomerId	NUMBER	6	PRIMARY KEY	
firstname	VARCHAR2	15	NOT NULL	
lastName	VARCHAR2	20	NOT NULL	
areaCode	CHAR	3		
telephoneNumber	CHAR	7		
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY REF Zips	
dateFilledIn	DATE			
preferredArtistId	NUMBER	6	FOREIGN KEY REF Artist	
preferredMedium	VARCHAR2	15		
preferredStyle	VARCHAR2	15		
preferredType	VARCHAR2	20		

TABLE Artwork

Item	Datatype	Size	Constraints	Comments
artworkId	NUMBER	6	PRIMARY KEY	
artistId	NUMBER	6	FOREIGN KEY REF Artist; NOT NULL; (artistId, workTitle) UNIQUE	
workTitle	VARCHAR2	50	NOT NULL; (artistId, workTitle) UNIQUE	
askingPrice	NUMBER	8,2		
dateListed	DATE			
dateReturned	DATE			
dateShown	DATE			
status	VARCHAR2	15		
workMedium	VARCHAR2	15		
workSize	VARCHAR2	15		
workStyle	VARCHAR2	15		
workType	VARCHAR2	20		
workYearCompleted	CHAR	4		
collectorSocialSecurity-Number	CHAR	9	FOREIGN KEY REF Collector	

TABLE Show

Item	Datatype	Size	Constraints	Comments
showTitle	VARCHAR2	50	PRIMARY KEY	
showFeaturedArtistId	NUMBER	6	FOREIGN KEY REF Arist	
showClosingDate	DATE			
showTheme	VARCHAR2	50		
showOpeningDate	DATE			

TABLE ShownIn

Item	Datatype	Size	Constraints	Comments
artworkId	NUMBER	6	PRIMARY KEY(artworkId, showTitle); FOREIGN KEY REFArtwork	
showTitle	VARCHAR2	50	PRIMARY KEY(artworkId, showTitle); FOREIGN KEY REF Show	

TABLE Buyer

buyerId	NUMBER6		PRIMARY KEY	
firstName	VARCHAR2	15	NOT NULL	
lastName	VARCHAR2	20	NOT NULL	
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY REF Zips	
areaCode	CHAR	3		
telephoneNumber	CHAR	7		
purchasesLastYear	NUMBER	8,2		
purchasesYearToDate	NUMBER	8,2		

TABLE Salesperson

Item	Datatype	Size	Constraints	Comments
socialSecurityNumber	CHAR	9	PRIMARY KEY	
firstName	VARCHAR2	15	NOT NULL; (firstName,lastName) UNIQUE	
lastName	VARCHAR2	20	NOT NULL; (firstName,lastName) UNIQUE	
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY REF Zips	

TABLE Sale

Item	Datatype	Size	Constraints	Comments
invoiceNumber	NUMBER	6	PRIMARY KEY	
artworkId	NUMBER	6	NOT NULL; UNIQUE; FOREIGN KEY REF Artwork	
amountRemittedToOwner	NUMBER	8,2	DEFAULT 0.00	
saleDate	DATE			
salePrice	NUMBER	8,2		
saleTax	NUMBER	6,2		
buyerId	NUMBER	6	NOT NULL; FOREIGN KEY REF Buyer	
salespersonSocialSecurityNumber	CHAR	9		

- Step 6.2. Write and execute SQL statements to create all the tables needed to implement the design.

Because we wish to specify foreign keys as we create the tables, we must be careful of the order in which we create, because the "home table" has to exist before the table containing the foreign key is created. Therefore, we will use the following order: Zips, Artist, Collector, Potential Customer, Artwork, Show, ShownIn, Buyer, Salesperson, Sale. The DDL statements to create the tables are shown in Figure 6.9. We are using Oracle syntax, but the DDL statements should work, with minor modifications, for any relational DBMS.

- Step 6.3. Create indexes for foreign keys and any other columns that will be used most often for queries.

The DDL statements to create the indexes are shown in Figure 6.10.

- Step 6.4. Insert about five records in each table, preserving all constraints. Put in enough data to demonstrate how the database will function.

Figure 6.11 shows the INSERT statements. Because we wish to make use of Oracle's system-generated values for surrogate keys, we created sequences for each of artistId, potentialCustomerId, artworkId, and buyerId, using this command:

```
CREATE SEQUENCE sequence-name
[START WITH starting-value]
[INCREMENT BY step] . . .;
```

```
CREATE TABLE Zips (
        zip CHAR (5),
        city VARCHAR2 (15) NOT NULL,
        state CHAR (2) NOT NULL,
        CONSTRAINT Zips_zip_pk PRIMARY KEY (zip));

CREATE TABLE Artist (
        ArtistId NUMBER (6),
        firstName VARCHAR2 (15) NOT NULL,
        lastName VARCHAR2 (20) NOT NULL,
        interviewDate DATE,
        interviewerName VARCHAR2 (35),
        areaCode   CHAR (3),
        telephoneNumber   CHAR (7),
        street VARCHAR2 (50),
        zip   CHAR (5),
        salesLastYear NUMBER (8, 2),
        salesYearToDate    NUMBER (8, 2),
        socialSecurityNumber CHAR (9),
        usualMedium VARCHAR (15),
        usualStyle VARCHAR (15),
        usualType VARCHAR (20),
        CONSTRAINT Artist_ArtistId_pk PRIMARY KEY (ArtistId),
        CONSTRAINT Artist_socialSecurityNumber_uk UNIQUE (socialSecurityNumber),
        CONSTRAINT Artist_firstName_lastName_uk UNIQUE (firstName,   lastName),
        CONSTRAINT Artist_zip_fk FOREIGN KEY (zip) REFERENCES Zips (zip)) ON UPDATE
CASCADE ON DELETE SET NULL;

CREATE TABLE Collector (
        socialSecurityNumber CHAR (9),
        firstName VARCHAR2 (15) NOT NULL,
        lastName VARCHAR2 (20) NOT NULL,
        interviewDate DATE,
        interviewerName VARCHAR2 (35),
        areaCode CHAR (3),
        telephoneNumber    CHAR (7),
```

(continued)

FIGURE 6.9

Oracle DDL Statements for The Art Gallery Tables

```
        street VARCHAR2 (50),
        zip CHAR (5),
        salesLastYear NUMBER (8, 2),
        salesYearToDate     NUMBER (8, 2),
        collectionArtistId NUMBER (6),
        collectionMedium VARCHAR (15),
        collectionStyle VARCHAR (15),
        collectionType VARCHAR (20),
        CONSTRAINT Collector_socialSecurityNumber_pk PRIMARY KEY
        (socialSecurityNumber).
        CONSTRAINT Collector_collectionArtistid_fk FOREIGN KEY (collectionArtistId)
REFERENCES Artist (artistId) ON UPDATE CASCADE ON DELETE NO ACTION;
        CONSTRAINT Collector_zip_fk FOREIGN KEY (zip) REFERENCES Zips (zip)) ON UPDATE
CASCADE ON DELETE SET NULL;

CREATE TABLE PotentialCustomer (
        potentialCustomerId NUMBER (6),
        firstname VARCHAR2 (15) NOT NULL,
        lastName VARCHAR2 (20) NOT NULL,
        areaCode CHAR (3),
        telephoneNumber CHAR (7),
        street VARCHAR2 (50),
        zip CHAR (5),
        dateFilledIn DATE,
        preferredArtistId NUMBER (6),
        preferredMedium VARCHAR2 (15),
        preferredStyle VARCHAR2 (15),
        preferredType VARCHAR2 (20),
        CONSTRAINT PotentialCustomer_potentialCustomerId_pk PRIAMRY KEY
        (potentialCustomerId),
        CONSTRAINT PotentialCustomer_zip_fk FOREIGN KEY (zip)
        REFERENCES Zips (zip) ON UPDATE CASCADE ON DELETE SET NULL,
        CONSTRAINT Potential Customer preferredArtistId_fk FOREIGN KEY
        (preferredArtistId) REFERENCES Artist (artist_Id)) ON UPDATE CASCADE ON DELETE SET NULL;

CREATE TABLE Artwork (
        artworkId NUMBER (6),
        artistId NUMBER (6) NOT NULL,
```

FIGURE 6.9—Continued

```
        workTitle VARCHAR2 (50) NOT NULL,
        askingPrice NUMBER (8, 2),
        dateListed DATE,
        dateReturned DATE,
        dateShown DATE,
        status VARCHAR2 (15),
        workMedium VARCHAR2 (15),
        workSize VARCHAR2 (15),
        workStyle VARCHAR2 (15),
        workType VARCHAR2 (20),
        workYearCompeted CHAR (4),
        collectorSocialSecurityNumber CHAR (9),
        CONSTRAINT Artwork_artworkId_pk PRIMARY KEY (artworkId).
        CONSTRAINT Artwork_artistId_workTitle_uk UNIQUE (artistId, workTitle),
        CONSTRAINT Artwork_artistId_fk FOREIGN KEY (artistId) REFERENCES Artist (artistId)
        ON UPDATE CASCADE ON DELETE NO ACTION,
        CONSTRAINT Artwork_collectorSocialSecurityNumber_fk FOREIGN KEY
(collectorSocialSecurityNumber)   REFERENCES Collector (socialSecurityNumber) ON UPDATE
CASCADE ON DELETE NO ACTION);

CREATE TABLE Show (
        showTitle VARCHAR2 (50),
        showFeaturedArtistId NUMBER (6),
        showClosingDate DATE,
        showTheme VARCHAR2 (50),
        showOpeningDate DATE,
        CONSTRAINT Show_showTitle_pk PRIMARY KEY (showTitle),
        CONSTRAINT Show_showFeaturedArtistId_fk FOREIGN KEY (showFeaturedArtistId)
REFERENCES Artist (artistId) ON UPDATE CASCADE ON DELETE NO ACTION);

CREATE TABLE ShownIn (
        artworkId NUMBER (6),
        showTitle VARCHAR2 (50),
        CONSTRAINT ShownIn_artworkid_showTitle_pk PRIMARY KEY (artworkId, showTitle),
        CONSTRAINT ShownIn_artworkId_fk FOREIGN KEY (artworkId) REFERENCES Artwork
        (artworkId) ON UPDATE CASCADE ON DELETE NO ACTION,
```

(continued)

FIGURE 6.9—Continued

```
          CONSTRAINT ShownIn_showTitle_fk FOREIGN KEY (showTitle) REFERENCES Show
(showTitle) ON UPDATE CASCADE ON DELETE NO ACTION);
CREATE TABLE Buyer (
          buyerId NUMBER (6),
          firstName VARCHAR2 (15) NOT NULL,
          lastName VARCHAR2 (20) NOT NULL,
          street VARCHAR2 (50),
          zip CHAR (5),
          areaCode CHAR (3),
          telephoneNumber CHAR (7),
          purchasesLastYear NUMBER (8, 2),
          purchasesYearToDate NUMBER (8, 2),
          CONSTRAINT Buyer_buyerId_pk PRIMARY KEY (buyerId),
          CONSTRAINT Buyer_zip_fk FOREIGN KEY (zip) REFERENCES Zips (zip) ON UPDATE
          CASCADE ON DELETE SET NULL);
CREATE TABLE Salesperson (
          socialSecurityNumber CHAR (9),
          firstName VARCHAR2 (15) NOT NULL,
          lastName VARCHAR2 (20) NOT NULL,
          street VARCHAR2 (50),
          zip CHAR (5),
          CONSTRAINT Salesperson_socialSecurityNumber_pk PRIMARY KEY
          socialSecurityNumber),
          CONSTRAINT Salesperson_firstName_lastName_uk UNIQUE (firstName, lastName),
          CONSTRAINT Salesperson_zip_fk FOREIGN KEY (zip) REFERENCES Zips (zip) ON
          UPDATE
CASCADE ON DELETE SET NULL);
CREATE TABLE Sale (
          invoiceNumber NUMBER (6),
          artworkId NUMBER (6) NOT NULL,
          amountRemittedToOwner NUMBER (8, 2) DEFAULT 0.00,
          saleDate DATE,
          salePrice NUMBER (8, 2),
          saleTax NUMBER (6, 2),
          buyerId NUMBER (6) NOT NULL,
```

FIGURE 6.9—Continued

```
        salespersonSocialSecurityNumber CHAR (9),
        CONSTRAINT Sale_invoiceNumber_pk PRIMARY KEY (invoiceNumber),
        CONSTRAINT Sale_artworkId_uk UNIQUE (artworkId),
        CONSTRAINT Sale_artworkId_fk FOREIGN KEY (artworkId) REFERENCES Artwork (artworkId)
        ON UPDATE CASCADE ON DELETE NO ACTION,
        CONSTRAINT Sale_buyerId_fk FOREIGN KEY (buyerId) REFERENCES Buyer (buyerId) ON
        UPDATE CASCADE ON DELETE NO ACTION);
```

```
CREATE UNIQUE INDEX Artist_lastName_firstName ON Artist(lastName, firstName);
CREATE UNIQUE INDEX Artist_socialSecurityNumber ON Artist(socialSecurityNumber);
CREATE INDEX Artist_zip ON Artist(zip);

CREATE INDEX Collector_collectionArtistId On Collector(collectionArtistId);
CREATE INDEX Collector_zip ON Collector(zip);
CREATE INDEX Collector_lastName_firstName ON Collector(lastName, firstName);

CREATE INDEX PotentialCustomer_zip ON PotentialCustomer(zip);
CREATE INDEX PotentialCustomer_preferredArtistId ON PotentialCustomer(preferredArtistId);
CREATE INDEX PotentialCustomer_lastName_firstName ON PotentialCustomer(lastName,
firstName);

CREATE UNIQUE INDEX Artwork_artistId_workTitle ON Artwork (artistId, workTitle);
CREATE INDEX Artwork_artistId ON Artwork(artistId);
CREATE INDEX Artwork_collectorSocialSecurityNumber ON Artwork
(collectorSocial-SecurityNumber);

CREATE INDEX Show_showFeaturedArtistId On Show (showFeaturedArtistId);

CREATE INDEX Shownin_artworkId ON Shownin (artworkId);
CREATE INDEX Shownin_show Title ON ShownIn (showTitle);

CREATE INDEX Buyer_zip ON Buyer(zip);
CREATE INDEX Buyer_lastName_firstName ON Buyer (lastName, firstName);

CREATE UNIQUE INDEX Salesperson_lastName_firstName ON Salesperson (lastName, firstName);
CREATE INDEX Salesperson_zip ON Salespeson (zip);

CREATE UNIQUE INDEX Sale_artworkId ON Sale (artworkId);
CREATE INDEX Sale_buyerId ON Sale (buyerId);
```

FIGURE 6.10
Oracle DDL Statements for The Art Gallery Indexes

```
INSERT INTO Zips VALUES ('10101,''New York,''NY');
INSERT INTO Zips VALUES ('10801','New Rochelle,''NY');
INSERT INTO Zips VALUES ('92101','San Diego','CA');
INSERT INTO Zips VALUES ('33010','Miami','FL');
INSERT INTO Zips VALUES ('60601','Chicago','IL');

CREATE SEQUENCE artistId sequence;
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL,'Leonardo','Vincenti','10-Oct-1999',
'Hughes,''212','5559999','10 Main Street,''10101',9000,4500,'099999876','oil','realism','painting');
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL,'Vincent','Gogh','15-Jun-2004',
'Hughes','914,''5551234,''55 West 18 Street','10801',9500,5500,'099999877','oil',
'impressionism','painting');
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL,'Winslow','Homes','05-Jan-2004',
'Hughes','619','1234567','100 Water Street','92101',14000,4000,'083999876','watercolor',
'realism','painting');
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL,'Alexander','Calderone','10-Feb-
1999','Hughes','212','5559999','10 Main Street','10101',20000,20000,'123999876','steel',
'cubism','sculpture');
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL,'Georgia','Keefe','05-Oct-2004',
'Hughes','305','1239999','5 Chestnut Street','33010',19000,14500,'987999876','oil','realism',
'painting');

INSERT INTO Collector VALUES('102345678','John','Jackson','01-Feb-2004','Hughes','917',
'7771234','24 Pine Avenue,''10101',4000,3000,1,'oil','realism','collage');
INSERT INTO Collector VALUES ('987654321','Mary','Lee','01-Mar-2003','Jones','305',
'5551234','10 Ash Street',33010,'2000',3000,2,'watercolor','realism','painting');
INSERT INTO Collector VALUES('034345678','Ramon','Perez','15-Apr-2003','Hughes','619',
'8881234','15 Poplar Avenue','92101',4500,3500,3,'oil','realism','painting');
INSERT INTO Collector VALUES('888881234','Rick','Lee','20-Jun-2004','Hughes','212',
'9991234','24 Pine Avenue','10101',4000,3000,3,'oil','realism','sculpture');
INSERT INTO Collector VALUES('777345678','Samantha','Torno','05-May-2004','Jones','305',
'5551234','10 Ash Street','33010',40000,30000,1,'acrylic','realism','painting');

CREATE SEQUENCE potentialCustomerId-sequence;
INSERT INTO PotentialCustomer VALUES(potentialCustomerId_sequence.NEXTVAL,'Adam',
'Burns','917','3456789','1 Spruce Street','10101','12-Dec-2003',1,'watercolor','impressionism',
'painting');
INSERT INTO PotentialCustomer VALUES(potentialCustomerId sequence.NEXTVAL,'Carole','Burns',
'917','3456789','1 Spruce Street','10101','12-Dec-2003',2,'watercolor','realism',sculpture');
INSERT INTO PotentialCustomer VALUES(potentialCustomerId_sequence.NEXTVAL,'David',
'Engel','914','7777777','715 North Avenue','10801','08-Aug-2003',3,'watercolor','realism',
'painting');
```

FIGURE 6.11

INSERT statements to populate The Art Gallery Tables

```
INSERT INTO PotentialCustomer VALUES(potentialCustomerId_sequence.NEXTVAL,‘Frances’,
‘Hughes’,‘619’,‘3216789’,‘10 Pacific Avenue’,‘92101’,‘05-Jan-2004’, 2,‘oil’,‘impressionism’,
‘painting’);
INSERT INTO PotentialCustomer VALUES(potentialCustomerId_sequence.NEXTVAL,‘Irene’,
‘Jacobs’,‘312’,‘1239876’,‘1 Windswept Place’,‘60601’,‘21-Sep-2003’, 5,‘watercolor’,‘abstract
expressionism’,‘painting’);

CREATE SEQUENCE artworkId_sequence;
INSERT INTO Artwork VALUES(artworkId_sequence.NEXTVAL, 1,‘Flight’, 15000.00,‘08-Sep-
2003’,,,‘for sale’,‘oil’,‘36 in × 48 in’,‘realism’,‘painting,’‘2001’,);
INSERT INTO Artwork VALUES(artworkId_sequence.NEXTVAL, 3,‘Bermuda Sunset’, 8000.00,
‘15-Mar-2004’, ,‘01-Apr-2004’,‘solid’,‘watercolor’,‘22 in × 28 in’,‘realism’,‘painting’, 2003’,);
INSERT INTO Artwork VALUES(artworkId_sequence.NEXTVAL, 3,‘Mediterranean Coast’,
4000.00,‘18-Oct-2003’, ,‘01-Apr-2004’,‘for sale’,‘watercolor’,‘22 in × 28 in’,‘realism’,‘paint-
ing,’‘2000’,‘102345678’);
INSERT INTO Artwork VALUES(artworkId_sequence.NEXTVAL, 5,‘Ghost orchid’, 18000.00,‘05-
Jun-2003’, , ,‘sold’,‘oil’,‘36 in × 48 in’,‘realism’,‘painting’,‘2001’,‘034345678’);
INSERT INTO Artwork VALUES(artworkId_sequence.NEXTVAL, 4,‘Five Planes’, 15000.00,‘10-
Jan-2004’, ,‘10-Mar-2004’‘for sale’,‘steel’,‘36 in X 60 in’,‘cubism’,‘sculpture’,‘2003’,
‘034345678’);

INSERT INTO Show VALUES(‘The Sea in Watercolor’, 3,‘30-Apr-2004’,‘seascapes’,‘01-Apr-2004’);
INSERT INTO Show VALUES(‘Calderone’’s Mastery of Space’, 4,‘20-Mar-2004’, ,‘10-Mar-2004’);

INSERT INTO Shownin VALUES(2,‘The Sea in Watercolor’);
INSERT INTO Shownin VALUES(3,‘The Sea in Watercolor’);
INSERT INTO Shownin VALUES(5,‘Calderone’’s Mastery of Space’);

CREATE SEQUENCE buyerId_sequence;
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL,‘Valerie’,‘Smiley’,‘15 Hudson
Street’,‘10101’,‘718’,‘5551234’, 5000, 7500);
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL,‘Winston’,‘Lee’,‘20 Liffey Avenue’,
‘60601’,‘312’,‘7654321’, 3000, 0);
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL,‘Samantha’,‘Babson’,‘25 Thames
Lane’,‘92101’,‘619’,‘4329876’, 15000, 0);
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL,‘John’,‘Flagg’,‘22 Amazon Street’,
‘10101’,‘212’,‘7659876’, 3000, 0);
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL,‘Terrence’,‘Smallshaw’,‘5 Nile
Street’,‘33010’,‘305’,‘2323456’, 15000, 17000);

INSERT INTO Salesperson VALUES(‘102445566’,‘John’,‘Smith’,‘10 Sapphire Row’,‘10801’);
INSERT INTO Salesperon VALUES(‘121344321’,‘Alan’,‘Hughes’,‘10 Diamond Street’,‘10101’);
INSERT INTO Salesperson VALUES(‘101889988’,‘Mary’,‘Brady’,‘10 Pearl Avenue’,‘10801’);
INSERT INTO Salesperson VALUES(‘111223344’,‘Jill’,‘Fleming’,‘10 Ruby Row’,‘10101’);
INSERT INTO Salesperson VALUES(‘123123123’,‘Terrence’,‘DeSimone’,‘10 Emerald Lane’,‘10101’);

INSERT INTO Sale VALUES(1234, 2,‘05-Apr-2004’, 7500, 600, 1,‘102445566’);
INSERT INTO Sale VALUES(1235, 4, ,‘06-Apr-2004’, 17000, 1360, 5,‘121344321’);
```

We chose to start with one and increment by one, the defaults. To generate each new value, we use the command <*sequence_name*>.NEXTVAL, as shown in the INSERT commands. We assume `invoiceNumber` is a number preprinted on the invoice form, not system-generated, so we do not need a sequence for that number.

- **Step 6.5.** Write SQL statements that will process five non-routine requests for information from the database just created. For each, write the request in English, followed by the corresponding SQL command.

1. Find the names of all artists who were interviewed after January 1, 2004, but who have no works of art listed.

```
SELECT firstName, lastName
FROM Artist
WHERE interviewDate > 01-Jan-2004 AND NOT EXISTS
   (SELECT *
   FROM Artwork
   WHERE artistId =Artist.artistId);
```

2. Find the total commission for salesperson John Smith earned between the dates April 1, 2004, and April 15, 2004. Recall that the gallery charges 10% commission, and the salesperson receives one-half of that, which is 5% of the selling price.

```
SELECT .05 * SUM(salePrice)
FROM Sale
WHERE saleDate > = 01-Apr-2004 AND
   saleDate < = 15-Apr-2004 AND
   salespersonSocialSecurityNumber = (SELECT socialSecurityNumber
   FROM Salesperson
   WHERE firstName= 'John' AND lastName ='Smith');
```

3. Find the collector names, artist names, and titles of all artworks that are owned by collectors, not by the artists themselves, in order by the collector's last name.

```
SELECT Collector.firstName, Collector.lastName,
Artist.firstName, Artist.lastName, workTitle
FROM Artist, Artwork, Collector
WHERE Artist.artistId = Artwork.artistId AND
Artwork.collectorSocialSecurityNumber =
Collector.socialSecurityNumber AND
collectorSocialSecurityNumber IS NOT NULL
ORDER BY Collector.lastName, Collector.firstName;
```

4. For each potential buyer, find information about shows that feature his or her preferred artist.

```
SELECT firstName, lastName, showTitle, showOpeningDate,
showClosingDate
FROM Show, PotentialCustomer
WHERE showFeaturedArtistId = PotentialCustomer.preferredArtistId
GROUP BY potentialCustomerId;
```

5. Find the average sale price of works of artist Georgia Keefe.

```
SELECT AVG(salePrice)
FROM Sale
WHERE artworkId IN (SELECT artworkId
   FROM Artwork
   WHERE artistId = (SELECT ArtistId
                     FROM Artist
                     WHERE lastName = 'Keefe' AND firstName
                     ='Georgia'));
```

- **Step 6.6.** Create at least one trigger and write the code for it. This trigger will update the amount of the buyer's purchases year-to-date whenever a sale is completed.

```
CREATE TRIGGER UPDATEBUYERYTD
AFTER INSERT ON Sale
   UPDATE Buyer
   SET purchasesYearToDate = purchasesYearToDate + :NEW.salePrice
   WHERE Buyer.buyerId = :NEW.buyerId;
```

STUDENT PROJECTS: CREATING AND USING A RELATIONAL DATABASE FOR THE STUDENT PROJECT

For the normalized tables you developed at the end of Chapter 5 for the project you have chosen, carry out the following steps to implement the design using a relational database management system such as Oracle, SQLServer, or MySQL.

- Step 6.1. Update the data dictionary and list of assumptions as needed. For each table, write the table name and write out the names, data types, and sizes of all the data items, and identify any constraints, using the conventions of the DBMS you will use for implementation.
- Step 6.2. Write and execute SQL statements to create all tables needed to implement the design.

- Step 6.3. Create indexes for foreign keys and for any other columns as needed.
- Step 6.4. Insert at least five records in each table, preserving all constraints. Put in enough data to demonstrate how the database will function.
- Step 6.5. Write SQL statements that will process five non-routine requests for information from the database just created. For each, write the request in English, followed by the corresponding SQL command.
- Step 6.6. Create at least one trigger and write the code for it.