In the previous chapter, we described several aspects of the SQL language, the standard for relational databases. We described the SQL statements for data definition, schema modification, queries, and updates. We also described how common constraints such as key and referential integrity are specified. In this chapter, we present several additional aspects of SQL. We start in Section 9.1 by describing the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. Then, in Section 9.2, we describe the SQL facilities for defining views on the database. Views are also called virtual or derived tables because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables.

The next several sections of this chapter discuss various techniques for accessing databases from programs. Most database access in practical situations is through software programs that implement database applications. This software is usually developed in a general-purpose programming language such as JAVA, COBOL, or C/C++. Recall from Section 2.3.1 that when database statements are included in a program, the general-purpose programming language is called the host language, whereas the database language—SQL, in our case—is called the data sublanguage. In some cases, special database programming languages are developed specifically for writing database applications. Although many of these were developed as research prototypes, some notable database programming languages have widespread use, such as ORACLE’s PL/SQL (Programming Language/SQL).
We start our presentation of database programming in Section 9.3 with an overview of the different techniques developed for accessing a database from programs. Then, in Section 9.4, we discuss the rules for embedding SQL statements into a general-purpose programming language, generally known as embedded SQL. This section also briefly discusses dynamic SQL, in which queries can be dynamically constructed at runtime, and presents the basics of the SQLJ variation of embedded SQL that was developed specifically for the programming language JAVA. In Section 9.5, we discuss the technique known as SQL/CLI (Call Level Interface), in which a library of procedures and functions is provided for accessing the database. Various sets of library functions have been proposed. The SQL/CLI set of functions is the one given in the SQL standard. Another library of functions is ODBC (Open Data Base Connectivity). We do not describe ODBC because it is considered to be the predecessor to SQL/CLI. A third library of functions—which we do describe—is JDBC; this was developed specifically for accessing databases from JAVA. Finally, in Section 9.6, we discuss SQL/PSM (Persistent Stored Modules), which is a part of the SQL standard that allows program modules—procedures and functions—to be stored by the DBMS and accessed through SQL. Section 9.7 summarizes the chapter.

9.1 SPECIFYING GENERAL CONSTRAINTS AS ASSERTIONS

In SQL, users can specify general constraints—those that do not fall into any of the categories described in Section 8.2—via declarative assertions, using the CREATE ASSERTION statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query. For example, to specify the constraint that “the salary of an employee must not be greater than the salary of the manager of the department that the employee works for” in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS
     (SELECT *
      FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
      WHERE E.SALARY>M.SALARY AND
          E.DNO=D.DNUMBER AND
          D.MGRSSN=M.SSN)  );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a condition in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to refer to the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions. Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint isviolated. The constraint is satisfied by a database state if no combination of tuples in that database state violates the constraint.
The basic technique for writing such assertions is to specify a query that selects any
tuples that violate the desired condition. By including this query inside a NOT EXISTS clause,
the assertion will specify that the result of this query must be empty. Thus, the assertion is
violated if the result of the query is not empty. In our example, the query selects all
employees whose salaries are greater than the salary of the manager of their department. If
the result of the query is not empty, the assertion is violated.

Note that the CHECK clause and constraint condition can also be used to specify
constraints on attributes and domains (see Section 8.2.1) and on tuples (see Section
8.2.4). A major difference between CREATE ASSERTION and the other two is that the
CHECK clauses on attributes, domains, and tuples are checked in SQL only when tuples are
inserted or updated. Hence, constraint checking can be implemented more efficiently by
the DBMS in these cases. The schema designer should use CHECK on attributes, domains,
and tuples only when he or she is sure that the constraint can only be violated by insertion
or updating of tuples. On the other hand, the schema designer should use CREATE
ASSERTION only in cases where it is not possible to use CHECK on attributes, domains,
or tuples, so that checks are implemented more efficiently by the DBMS.

Another statement related to CREATE ASSERTION in SQL is CREATE TRIGGER, but
triggers are used in a different way. In many cases it is convenient to specify the type of
action to be taken when certain events occur and when certain conditions are satisfied.
Rather than offering users only the option of aborting an operation that causes a
violation—as with CREATE ASSERTION—the DBMS should make other options available.
For example, it may be useful to specify a condition that, if violated, causes some user to
be informed of the violation. A manager may want to be informed if an employee’s travel
expenses exceed a certain limit by receiving a message whenever this occurs. The action
that the DBMS must take in this case is to send an appropriate message to that user. The
condition is thus used to monitor the database. Other actions may be specified, such as
executing a specific stored procedure or triggering other updates. The CREATE TRIGGER
statement is used to implement such actions in SQL. A trigger specifies an event (such as
a particular database update operation), a condition, and an action. The action is to be executed automatically if the condition is satisfied when the event occurs. We discuss
triggers in detail in Section 24.1 when we describe active databases.

9.2 Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL. We then show how views are
specified, and we discuss the problem of updating a view, and how a view can be imple-
mented by the DBMS.

9.2.1 Concept of a View in SQL

A view in SQL terminology is a single table that is derived from other tables. These other
tables could be base tables or previously defined views. A view does not necessarily exist in

---

1. As used in SQL, the term view is more limited than the term user view discussed in Chapters 1 and
2, since a user view would possibly include many relations.
physical form; it is considered a virtual table, in contrast to base tables, whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically. For example, in Figure 5.5 we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the EMPLOYEE, WORKS_ON, and PROJECT tables every time we issue that query, we can define a view that is a result of these joins. We can then issue queries on the view, which are specified as single-table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE, WORKS_ON, and PROJECT tables the **defining tables** of the view.

### 9.2.2 Specification of Views in SQL

In SQL, the command to specify a view is `CREATE VIEW`. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case. The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 9.1 when applied to the database schema of Figure 5.5.

| V1: CREATE VIEW WORKS_ON1 AS SELECT FNAME, LNAME, PNAME, HOURS FROM EMPLOYEE, PROJECT, WORKS_ON WHERE SSN=ESSN AND PNO=PNUMBER; |  |
| V2: CREATE VIEW DEPT_INFO(DEPT_NAME, NO_OF_EMPS, TOTAL_SAL) AS SELECT DNAME, COUNT (*), SUM (SALARY) FROM DEPARTMENT, EMPLOYEE WHERE DNUMBER=DNO GROUP BY DNAME; |  |

<table>
<thead>
<tr>
<th>WORKS_ON1</th>
<th>WORKS_ON1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNAME</td>
<td>LNAME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPT_INFO</th>
<th>DEPT_INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPT_NAME</td>
<td>NO_OF_EMPS</td>
</tr>
</tbody>
</table>

**Figure 9.1** Two views specified on the database schema of Figure 5.5
9.2 Views (Virtual Tables) in SQL

In V1, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case, WORKS_ON1 inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON. View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables. For example, to retrieve the last name and first name of all employees who work on 'ProjectX', we can utilize the WORKS_ON1 view and specify the query as in QV1:

\[
QV1: \text{SELECT FNAME, LNAME FROM WORKS_ON1 WHERE PNAME='ProjectX';}
\]

The same query would require the specification of two joins if specified on the base relations; one of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism (see Chapter 23).

A view is supposed to be always up to date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view is not realized at the time of view definition but rather at the time we specify a query on the view. It is the responsibility of the DBMS and not the user to make sure that the view is up to date.

If we do not need a view any more, we can use the DROP VIEW command to dispose of it. For example, to get rid of the view V1, we can use the SQL statement in V1A:

\[
V1A: \text{DROP VIEW WORKS_ON1;}
\]

### 9.2.3 View Implementation and View Update

The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested. One strategy, called query modification, involves modifying the view query into a query on the underlying base tables. For example, the query QV1 would be automatically modified to the following query by the DBMS:

\[
\text{SELECT FNAME, LNAME FROM EMPLOYEE, PROJECT, WORKS_ON WHERE SSN=ESSN AND PNO=PNUMBER AND PNAME='ProjectX';}
\]

The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are applied to the view within a short period of time. The other strategy, called view materialization, involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when
the base tables are updated must be developed in order to keep the view up to date. Techniques using the concept of **incremental update** have been developed for this purpose, where it is determined what new tuples must be inserted, deleted, or modified in a materialized view table when a change is applied to one of the defining base tables. The view is generally kept as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical view table and recompute it from scratch when future queries reference the view.

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways. To illustrate potential problems with updating a view defined on multiple tables, consider the \texttt{WORKS\_ON1} view, and suppose that we issue the command to update the \texttt{PNAME} attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

\texttt{UV1: UPDATE WORKS\_ON1}
\texttt{SET PNAME = 'ProductY'}
\texttt{WHERE LNAME='Smith' AND FNAME='John' AND PNAME='ProductX';}

This query can be mapped into several updates on the base relations to give the desired update effect on the view. Two possible updates, (a) and (b), on the base relations corresponding to UV1 are shown here:

(a): \texttt{UPDATE WORKS\_ON}
\texttt{SET PNO = (SELECT PNUMBER FROM PROJECT WHERE PNAME='ProductY')}
\texttt{WHERE ESSN IN (SELECT SSN FROM EMPLOYEE WHERE LNAME='Smith' AND FNAME='John')}
\texttt{AND PNO = (SELECT PNUMBER FROM PROJECT WHERE PNAME='ProductX');}

(b): \texttt{UPDATE PROJECT SET PNAME = 'ProductY'}
\texttt{WHERE PNAME = 'ProductX';}

Update (a) relates 'John Smith' to the 'ProductY' \texttt{PROJECT} tuple in place of the 'ProductX' \texttt{PROJECT} tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the \texttt{PROJECT} relation to 'ProductY'. It is quite unlikely that the
user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with \textit{PNAME} = 'ProductX'.

Some view updates may not make much sense; for example, modifying the \textit{TOTAL_SAL} attribute of the \textit{DEPT_INFO} view does not make sense because \textit{TOTAL_SAL} is defined to be the sum of the individual employee salaries. This request is shown as UV2:

\begin{verbatim}
UV2: UPDATE DEPT_INFO
    SET TOTAL_SAL=100000
    WHERE DNAME='Research';
\end{verbatim}

A large number of updates on the underlying base relations can satisfy this view update.

A view update is feasible when only one possible update on the base relations can accomplish the desired update effect on the view. Whenever an update on the view can be mapped to more than one update on the underlying base relations, we must have a certain procedure for choosing the desired update. Some researchers have developed methods for choosing the most likely update, while other researchers prefer to have the user choose the desired update mapping during view definition.

In summary, we can make the following observations:

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the \textit{NOT NULL} constraint that do not have default values specified.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

In SQL, the clause \textit{WITH CHECK OPTION} must be added at the end of the view definition if a view is to be updated. This allows the system to check for view updatability and to plan an execution strategy for view updates.

\section*{9.3 Database Programming: Issues and Techniques}

We now turn our attention to the techniques that have been developed for accessing databases from programs and, in particular, to the issue of how to access SQL databases from application programs. Our presentation of SQL so far has focused on the language constructs for various database operations—from schema definition and constraint specification to querying, to updating, and the specification of views. Most database systems have an \textit{interactive interface} where these SQL commands can be typed directly into a monitor and input to the database system. For example, in a computer system where the \textsc{Oracle} RDBMS is installed, the command SQLPLUS will start the interactive interface. The user can type SQL commands or queries directly over several lines, ended by a semicolon and the Enter key (that is, “; <cr>”). Alternatively, a file of commands can be created and executed through the interactive interface by typing @<filename>. The system will execute the commands written in the file and display the results, if any.
Chapter 9  More SQL: Assertions, Views, and Programming Techniques

The interactive interface is quite convenient for schema and constraint creation or for occasional ad hoc queries. However, the majority of database interactions in practice are executed through programs that have been carefully designed and tested. These programs are generally known as **application programs** or **database applications**, and are used as canned transactions by the end users, as discussed in Section 1.4.3. Another very common use of database programming is to access a database through an application program that implements a **Web interface**, for example, for making airline reservations or department store purchases. In fact, the vast majority of Web electronic commerce applications include some database access commands.

In this section, we first give an overview of the main approaches to database programming. Then we discuss some of the problems that occur when trying to access a database from a general-purpose programming language, and discuss the typical sequence of commands for interacting with a database from a software program.

### 9.3.1 Approaches to Database Programming

Several techniques exist for including database interactions in application programs. The main approaches for database programming are the following:

1. **Embedding database commands in a general-purpose programming language**: In this approach, database statements are *embedded* into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string `EXEC SQL`, which precedes all SQL commands in a host language program. A **precompiler** or **preprocessor** first scans the source program code to identify database statements and extract them for processing by the DBMS. They are replaced in the program by function calls to the DBMS-generated code.

2. **Using a library of database functions**: A **library of functions** is made available to the host programming language for database calls. For example, there could be functions to connect to a database, execute a query, execute an update, and so on. The actual database query and update commands, and any other necessary information, are included as parameters in the function calls. This approach provides what is known as an **Application Programming Interface (API)** for accessing a database from application programs.

3. **Designing a brand-new language**: A **database programming language** is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional statements are added to the database language to convert it into a full-fledged programming language.

In practice, the first two approaches are more common, since many applications are already written in general-purpose programming languages but require some database access. The third approach is more appropriate for applications that have intensive database interaction. One of the main problems with the first two approaches is **impedance mismatch**, which does not occur in the third approach. We discuss this next.

---

2. Other prefixes are sometimes used, but this is the most common one.
9.3.2 Impedance Mismatch

Impedance mismatch is the term used to refer to the problems that occur because of differences between the database model and the programming language model. For example, the practical relational model has three main constructs: attributes and their data types, tuples (records), and tables (sets or multisets of records). The first problem that may occur is that the data types of the programming language differ from the attribute data types in the data model. Hence, it is necessary to have a binding for each host programming language that specifies for each attribute type the compatible programming language types. It is necessary to have a binding for each programming language because different languages have different data types; for example, the data types available in C and JAVA are different, and both differ from the SQL data types.

Another problem occurs because the results of most queries are sets or multisets of tuples, and each tuple is formed of a sequence of attribute values. In the program, it is often necessary to access the individual data values within individual tuples for printing or processing. Hence, a binding is needed to map the query result data structure, which is a table, to an appropriate data structure in the programming language. A mechanism is needed to loop over the tuples in a query result in order to access a single tuple at a time and to extract individual values from the tuple. A cursor or iterator variable is used to loop over the tuples in a query result. Individual values within each tuple are typically extracted into distinct program variables of the appropriate type.

Impedance mismatch is less of a problem when a special database programming language is designed that uses the same data model and data types as the database model. One example of such a language is ORACLE’s PL/SQL. For object databases, the object data model (see Chapter 20) is quite similar to the data model of the JAVA programming language, so the impedance mismatch is greatly reduced when JAVA is used as the host language for accessing a JAVA-compatible object database. Several database programming languages have been implemented as research prototypes (see bibliographic notes).

9.3.3 Typical Sequence of Interaction in Database Programming

When a programmer or software engineer writes a program that requires access to a database, it is quite common for the program to be running on one computer system while the database is installed on another. Recall from Section 2.5 that a common architecture for database access is the client/server model, where a client program handles the logic of a software application, but includes some calls to one or more database servers to access or update the data. When writing such a program, a common sequence of interaction is the following:

1. When the client program requires access to a particular database, the program must first establish or open a connection to the database server. Typically, this

---
3. As we discussed in Section 2.5, there are two-tier and three-tier architectures; to keep our discussion simple, we will assume a two-tier client/server architecture here. We discuss additional variations of these architectures in Chapter 25.
involves specifying the Internet address (URL) of the machine where the database server is located, plus providing a login account name and password for database access.

2. Once the connection is established, the program can interact with the database by submitting queries, updates, and other database commands. In general, most types of SQL statements can be included in an application program.

3. When the program no longer needs access to a particular database, it should terminate or close the connection to the database.

A program can access multiple databases if needed. In some database programming approaches, only one connection can be active at a time, whereas in other approaches multiple connections can be established at the same time.

In the next three sections, we discuss examples of each of the three approaches to database programming. Section 9.4 describes how SQL is embedded into a programming language. Section 9.5 discusses how function calls are used to access the database, and Section 9.6 discusses an extension to SQL called SQL/PSM that allows general-purpose programming constructs for defining modules (procedures and functions) that are stored within the database system.

9.4 EMBEDDED SQL, DYNAMIC SQL, AND SQLJ

9.4.1 Retrieving Single Tuples with Embedded SQL

In this section, we give an overview of how SQL statements can be embedded in a general-purpose programming language such as C, ADA, COBOL, or PASCAL. The programming language is called the host language. Most SQL statements—including data or constraint definitions, queries, updates, or view definitions—can be embedded in a host language program. An embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a preprocessor (or precompiler) can separate embedded SQL statements from the host language code. The SQL statements can be terminated by a semicolon (;) or a matching END-EXEC.

To illustrate the concepts of embedded SQL, we will use C as the host programming language. Within an embedded SQL command, we may refer to specially declared C program variables. These are called shared variables because they are used in both the C program and the embedded SQL statements. Shared variables are prefixed by a colon (:) when they appear in an SQL statement. This distinguishes program variable names from the names of database schema constructs such as attributes and relations. It also allows program variables to have the same names as attribute names, since they are distinguishable by the “:” prefix in the SQL statement.

4. Although SQL/PSM is not considered to be a full-fledged programming language, it illustrates how typical general-purpose programming constructs—such as loops and conditional structures—can be incorporated into SQL.
Names of database schema constructs—such as attributes and relations—can only be used within the SQL commands, but shared program variables can be used elsewhere in the C program without the “:" prefix.

Suppose that we want to write C programs to process the COMPANY database of Figure 5.5. We need to declare program variables to match the types of the database attributes that the program will process. The programmer can choose the names of the program variables; they may or may not have names that are identical to their corresponding attributes. We will use the C program variables declared in Figure 9.2 for all our examples, and we will show C program segments without variable declarations. Shared variables are declared within a declare section in the program, as shown in Figure 9.2 (lines 1 through 7). A few of the common bindings of C types to SQL types are as follows. The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C types long, short, float, and double, respectively. Fixed-length and varying-length strings (CHAR[i], VARCHAR[i]) in SQL can be mapped to arrays of characters (char [i+1], varchar [i+1]) in C that are one character longer than the SQL type, because strings in C are terminated by a "\0" (null) character, which is not part of the character string itself.

Notice that the only embedded SQL commands in Figure 9.2 are lines 1 and 7, which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE because they can be included in embedded SQL statements—as long as they are preceded by a colon (:). Lines 2 through 5 are regular C program declarations. The C program variables declared in lines 2 through 5 correspond to the attributes of the EMPLOYEE and DEPARTMENT tables from the COMPANY database of Figure 5.5 that was declared by the SQL DDL in Figure 8.1. The variables declared in line 6—SQLCODE and SQLSTATE—are used to communicate errors and exception conditions between the database system and the program. Line 0 shows a program variable loop that will not be used in any embedded SQL statements, so it is declared outside the SQL declare section.

```c
0) int loop ;
1) EXEC SQL BEGIN DECLARE SECTION ;
2) varchar dname [16], fname [16], lname [16], address [31] ;
3) char ssn [10], bdate [11], sex [2], minit [2] ;
4) float salary, raise ;
5) int dno, dnumber ;
6) int SQLCODE ; char SQLSTATE [6] ;
7) EXEC SQL END DECLARE SECTION ;
```

**FIGURE 9.2** C program variables used in the embedded SQL examples E1 and E2

5. We use line numbers in our code segments for easy reference; these numbers are not part of the actual code.
6. SQL strings can also be mapped to char* types in C.
Connecting to the Database. The SQL command for establishing a connection to a database has the following form:

```
CONNECT TO <server name> AS <connection name>
AUTHORIZATION <user account name and password> ;
```

In general, since a user or program can access several database servers, several connections can be established, but only one connection can be active at any point in time. The programmer or user can use the `<connection name>` to change from the currently active connection to a different one by using the following command:

```
SET CONNECTION <connection name> ;
```

Once a connection is no longer needed, it can be terminated by the following command:

```
DISCONNECT <connection name> ;
```

In the examples in this chapter, we assume that the appropriate connection has already been established to the `COMPANY` database, and that it is the currently active connection.

Communicating between the Program and the DBMS Using SQLCODE and SQLSTATE. The two special communication variables that are used by the DBMS to communicate exception or error conditions to the program are SQLCODE and SQLSTATE. The SQLCODE variable shown in Figure 9.2 is an integer variable. After each database command is executed, the DBMS returns a value in SQLCODE. A value of 0 indicates that the statement was executed successfully by the DBMS. If SQLCODE > 0 (or, more specifically, if SQLCODE = 100), this indicates that no more data (records) are available in a query result. If SQLCODE < 0, this indicates some error has occurred. In some systems—for example, in the ORACLE RDBMS—SQLCODE is a field in a record structure called SQLCA (SQL communication area), so it is referenced as SQLCA.SQLCODE. In this case, the definition of SQLCA must be included in the C program by including the following line:

```
EXEC SQL include SQLCA ;
```

In later versions of the SQL standard, a communication variable called SQLSTATE was added, which is a string of five characters. A value of "00000" in SQLSTATE indicates no error or exception; other values indicate various errors or exceptions. For example, "02000" indicates "no more data" when using SQLSTATE. Currently, both SQLSTATE and SQLCODE are available in the SQL standard. Many of the error and exception codes returned in SQLSTATE are supposed to be standardized for all SQL vendors and platforms, whereas the codes returned in SQLCODE are not standardized but are defined by the DBMS vendor. Hence, it is generally better to use SQLSTATE, because this makes error handling in the application programs independent of a particular DBMS. As an exercise, the reader should rewrite the examples given later in this chapter using SQLSTATE instead of SQLCODE.

---

7. In particular, SQLSTATE codes starting with the characters 0 through 4 or A through H are supposed to be standardized, whereas other values can be implementation-defined.
Example of Embedded SQL Programming. Our first example to illustrate embedded SQL programming is a repeating program segment (loop) that reads a social security number of an employee and prints out some information from the corresponding EMPLOYEE record in the database. The C program code is shown as program segment E1 in Figure 9.3. The program reads (inputs) a social security number value and then retrieves the EMPLOYEE tuple with that social security number from the database via the embedded SQL command. The INTO clause (line 5) specifies the program variables into which attribute values from the database are retrieved. C program variables in the INTO clause are prefixed with a colon (:), as we discussed earlier.

Line 7 in E1 illustrates the communication between the database and the program through the special variable SQLCODE. If the value returned by the DBMS in SQLCODE is 0, the previous statement was executed without errors or exception conditions. Line 7 checks this and assumes that if an error occurred, it was because no EMPLOYEE tuple existed with the given social security number; it therefore outputs a message to that effect (line 8).

In E1 a single tuple is selected by the embedded SQL query; that is why we are able to assign its attribute values directly to C program variables in the INTO clause in line 5. In general, an SQL query can retrieve many tuples. In that case, the C program will typically go through the retrieved tuples and process them one at a time. A cursor is used to allow tuple-at-a-time processing by the host language program. We describe cursors next.

9.4.2 Retrieving Multiple Tuples with Embedded SQL Using Cursors

We can think of a cursor as a pointer that points to a single tuple (row) from the result of a query that retrieves multiple tuples. The cursor is declared when the SQL query command is declared in the program. Later in the program, an OPEN CURSOR command fetches the query result from the database and sets the cursor to a position before the first row in the

```c
//Program Segment E1:
0) loop = 1;
1) while (loop) {
2) prompt("Enter a Social Security Number: ", ssn);
3) EXEC SQL
4) select FNAME, MINIT, LNAME, ADDRESS, SALARY
5) into :fname, :minit, :lname, :address, :salary
6) from EMPLOYEE where SSN = :ssn;
7) if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8) else printf("Social Security Number does not exist: ", ssn);
9) prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop);
10) }
```

FIGURE 9.3 Program segment E1, a C program segment with embedded SQL
result of the query. This becomes the current row for the cursor. Subsequently, FETCH commands are issued in the program; each FETCH moves the cursor to the next row in the result of the query, making it the current row and copying its attribute values into the C (host language) program variables specified in the FETCH command by an INTO clause. The cursor variable is basically an iterator that iterates (loops) over the tuples in the query result—one tuple at a time. This is similar to traditional record-at-a-time file processing.

To determine when all the tuples in the result of the query have been processed, the communication variable SQLCODE (or, alternatively, SQLSTATE) is checked. If a FETCH command is issued that results in moving the cursor past the last tuple in the result of the query, a positive value (SQLCODE > 0) is returned in SQLCODE, indicating that no data (tuple) was found (or the string “02000” is returned in SQLSTATE). The programmer uses this to terminate a loop over the tuples in the query result. In general, numerous cursors can be opened at the same time. A CLOSE CURSOR command is issued to indicate that we are done with processing the result of the query associated with that cursor.

An example of using cursors is shown in Figure 9.4, where a cursor called EMP is declared in line 4. We assume that appropriate C program variables have been declared as in Figure 9.2. The program segment in E2 reads (inputs) a department name (line 0), retrieves its department number (lines 1 to 3), and then retrieves the employees who

//Program Segment E2:
0) prompt(“Enter the Department Name: “, dname) ;
1) EXEC SQL
2) select DNUMBER into :dnumber
3) from DEPARTMENT where DNAME = :dname ;
4) EXEC SQL DECLARE EMP CURSOR FOR
5) select SSN, FNAME, MINIT, LNAME, SALARY
6) from EMPLOYEE where DNO = :dnumber
7) FOR UPDATE OF SALARY ;
8) EXEC SQL OPEN EMP ;
9) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE == 0) {
11) printf(“Employee name is: “, fname, minit, lname)
12) prompt(“Enter the raise amount: “, raise) ;
13) EXEC SQL
14) update EMPLOYEE
15) set SALARY = SALARY + :raise
16) where CURRENT OF EMP ;
17) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
18) }
19) EXEC SQL CLOSE EMP ;

FIGURE 9.4 Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes
work in that department via a cursor. A loop (lines 10 to 18) then iterates over each employee record, one at a time, and prints the employee name. The program then reads a raise amount for that employee (line 12) and updates the employee's salary in the database by the raise amount (lines 14 to 16).

When a cursor is defined for rows that are to be modified (updated), we must add the clause \texttt{FOR UPDATE} in the cursor declaration and list the names of any attributes that will be updated by the program. This is illustrated in line 7 of code segment E2. If rows are to be deleted, the keywords \texttt{FOR UPDATE} must be added without specifying any attributes. In the embedded \texttt{UPDATE} (or \texttt{DELETE}) command, the condition \texttt{WHERE CURRENT OF <cursor name>} specifies that the current tuple referenced by the cursor is the one to be updated (or deleted), as in line 16 of E2.

Notice that declaring a cursor and associating it with a query (lines 4 through 7 in E2) does not execute the query; the query is executed only when the \texttt{OPEN <cursor name>} command (line 8) is executed. Also notice that there is no need to include the \texttt{FOR UPDATE} clause in line 7 of E2 if the results of the query are to be used for retrieval purposes only (no update or delete).

Several options can be specified when declaring a cursor. The general form of a cursor declaration is as follows:

\begin{verbatim}
DECLARE <cursor name> [ INSENSITIVE ] [ SCROLL ] CURSOR
[ WITH HOLD ] FOR <query specification>
[ ORDER BY <ordering specification> ]
[ FOR READ ONLY | FOR UPDATE [ OF <attribute list> ] ] ;
\end{verbatim}

We already briefly discussed the options listed in the last line. The default is that the query is for retrieval purposes (\texttt{FOR READ ONLY}). If some of the tuples in the query result are to be updated, we need to specify \texttt{FOR UPDATE OF <attribute list> and list the attributes that may be updated. If some tuples are to be deleted, we need to specify \texttt{FOR UPDATE} without any attributes listed.

When the optional keyword \texttt{SCROLL} is specified in a cursor declaration, it is possible to position the cursor in other ways than for purely sequential access. A \texttt{fetch orientation} can be added to the \texttt{FETCH} command, whose value can be one of \texttt{NEXT}, \texttt{PRIOR}, \texttt{FIRST}, \texttt{LAST}, \texttt{ABSOLUTE i}, and \texttt{RELATIVE i}. In the latter two commands, \texttt{i} must evaluate to an integer value that specifies an absolute tuple position or a tuple position relative to the current cursor position, respectively. The default fetch orientation, which we used in our examples, is \texttt{NEXT}. The fetch orientation allows the programmer to move the cursor around the tuples in the query result with greater flexibility, providing random access by position or access in reverse order. When \texttt{SCROLL} is specified on the cursor, the general form of a \texttt{FETCH} command is as follows, with the parts in square brackets being optional:

\begin{verbatim}
FETCH [[ <fetch orientation> ] FROM] <cursor name> INTO <fetch target list> ;
\end{verbatim}

The \texttt{ORDER BY} clause orders the tuples so that the \texttt{FETCH} command will fetch them in the specified order. It is specified in a similar manner to the corresponding clause for \texttt{SQL} queries (see Section 8.4.6). The last two options when declaring a cursor (\texttt{INSENSITIVE} and \texttt{WITH HOLD}) refer to transaction characteristics of database programs, which we discuss in Chapter 17.
9.4.3 Specifying Queries at Runtime Using Dynamic SQL

In the previous examples, the embedded SQL queries were written as part of the host program source code. Hence, any time we want to write a different query, we must write a new program, and go through all the steps involved (compiling, debugging, testing, and so on). In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) dynamically at runtime. For example, we may want to write a program that accepts an SQL query typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs. Another example is when a user-friendly interface generates SQL queries dynamically for the user based on point-and-click operations on a graphical schema (for example, a QBE-like interface; see Appendix D). In this section, we give a brief overview of dynamic SQL, which is one technique for writing this type of database program, by giving a simple example to illustrate how dynamic SQL can work.

Program segment E3 in Figure 9.5 reads a string that is input by the user (that string should be an SQL update command) into the string variable sqlupdatestring in line 3. It then prepares this as an SQL command in line 4 by associating it with the SQL variable sqlcommand. Line 5 then executes the command. Notice that in this case no syntax check or other types of checks on the command are possible at compile time, since the command is not available until runtime. This contrasts with our previous examples of embedded SQL, where the query could be checked at compile time because its text was in the program source code.

Although including a dynamic update command is relatively straightforward in dynamic SQL, a dynamic query is much more complicated. This is because in the general case we do not know the type or the number of attributes to be retrieved by the SQL query when we are writing the program. A complex data structure is sometimes needed to allow for different numbers and types of attributes in the query result if no prior information is known about the dynamic query. Techniques similar to those that we discuss in Section 9.5 can be used to assign query results (and query parameters) to host program variables.

In E3, the reason for separating PREPARE and EXECUTE is that if the command is to be executed multiple times in a program, it can be prepared only once. Preparing the command generally involves syntax and other types of checks by the system, as well as

//Program Segment E3:
0) EXEC SQL BEGIN DECLARE SECTION ;
1) varchar sqlupdatestring [256] ;
2) EXEC SQL END DECLARE SECTION ;
...
3) prompt(“Enter the Update Command: ”, sqlupdatestring) ;
4) EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5) EXEC SQL EXECUTE sqlcommand ;
...

FIGURE 9.5 Program segment E3, a C program segment that uses dynamic SQL for updating a table
generating the code for executing it. It is possible to combine the PREPARE and EXECUTE
commands (lines 4 and 5 in E3) into a single statement by writing

    EXEC SQL EXECUTE IMMEDIATE :sqlupdatestring ;

This is useful if the command is to be executed only once. Alternatively, one can separate
the two to catch any errors after the PREPARE statement, if any.

9.4.4 SQLJ: Embedding SQL Commands in JAVA

In the previous sections, we gave an overview of how SQL commands can be embedded in
a traditional programming language, using the C language in our examples. We now turn
our attention to how SQL can be embedded in an object-oriented programming language,8
in particular, the JAVA language. SQLJ is a standard that has been adopted by several ven-
dors for embedding SQL in JAVA. Historically, SQLJ was developed after JDBC, which is
used for accessing SQL databases from JAVA using function calls. We discuss JDBC in Sec-
tion 9.5.2. In our discussion, we focus on SQLJ as it is used in the ORACLE RDBMS. An SQLJ
translator will generally convert SQL statements into JAVA, which can then be executed
through the JDBC interface. Hence, it is necessary to install a JDBC driver when using
SQLJ.9 In this section, we focus on how to use SQLJ concepts to write embedded SQL in a
JAVA program.

Before being able to process SQL with JAVA in ORACLE, it is necessary to import
several class libraries, shown in Figure 9.6. These include the JDBC and IO classes (lines 1
and 2), plus the additional classes listed in lines 3, 4, and 5. In addition, the program must
first connect to the desired database using the function call GETCONNECTION, which is one of
the methods of the ORACLE class in line 5 of Figure 9.6. The format of this function call,
which returns an object of type default context,10 is as follows:

    public static DefaultContext
    getConnection(String url, String user, String password, Boolean
    autoCommit)
    throws SQLException ;

For example, we can write the statements in lines 6 through 8 in Figure 9.6 to
connect to an ORACLE database located at the URL <url name> using the login of <user
name> and <password> with automatic commitment of each command,11 and then set
this connection as the default context for subsequent commands.

8. This section assumes familiarity with object-oriented concepts and basic JAVA concepts. If read-
ers lack this familiarity, they should postpone this section until after reading Chapter 20.
9. We discuss JDBC drivers in Section 9.5.2.
10. A default context, when set, applies to subsequent commands in the program until it is changed.
11. Automatic commitment roughly means that each command is applied to the database after it is
executed. The alternative is that the programmer wants to execute several related database com-
mands and then commit them together. We discuss commit concepts in Chapter 17 when we
describe database transactions.
In the following examples, we will not show complete JAVA classes or programs since it is not our intention to teach JAVA. Rather, we will show program segments that illustrate the use of SQLJ. Figure 9.7 shows the JAVA program variables used in our examples. Program segment J1 in Figure 9.8 reads an employee’s social security number and prints some of the employee’s information from the database.

Notice that because JAVA already uses the concept of exceptions for error handling, a special exception called SQLException is used to return errors or exception conditions after executing an SQL database command. This plays a similar role to SQLCODE and SQLSTATE in embedded SQL. JAVA has many types of predefined exceptions. Each JAVA operation (function) must specify the exceptions that can be thrown—that is, the exception conditions that may occur while executing the JAVA code of that operation. If a defined exception occurs, the system transfers control to the JAVA code specified for exception handling. In J1, exception handling for an SQLException is specified in lines 7 and 8.

Exceptions that can be thrown by the code in a particular operation should be specified as part of the operation declaration or interface—for example, in the following format:

```
<operation return type> <operation name>(<parameters>) throws SQLException, IOException;
```

In SQLJ, the embedded SQL commands within a JAVA program are preceded by #sql, as illustrated in J1 line 3, so that they can be identified by the preprocessor. SQLJ uses an INTO clause—similar to that used in embedded SQL—to return the attribute values retrieved from the database by an SQL query into JAVA program variables. The program variables are preceded by colons (:) in the SQL statement, as in embedded SQL.
In J1 a single tuple is selected by the embedded SQLJ query; that is why we are able to assign its attribute values directly to JAVA program variables in the INTO clause in line 4. For queries that retrieve many tuples, SQLJ uses the concept of an iterator, which is somewhat similar to a cursor in embedded SQL.

9.4.5 Retrieving Multiple Tuples in SQLJ Using Iterators

In SQLJ, an iterator is a type of object associated with a collection (set or multiset) of tuples in a query result. The iterator is associated with the tuples and attributes that appear in a query result. There are two types of iterators:

1. A named iterator is associated with a query result by listing the attribute names and types that appear in the query result.
2. A positional iterator lists only the attribute types that appear in the query result.

In both cases, the list should be in the same order as the attributes that are listed in the SELECT clause of the query. However, looping over a query result is different for the two types of iterators, as we shall see. First, we show an example of using a named iterator in Figure 9.9, program segment J2A. Line 9 in Figure 9.9 shows how a named iterator type Emp is declared. Notice that the names of the attributes in a named iterator type must match the names of the attributes in the SQL query result. Line 10 shows how an iterator object e of type Emp is created in the program and then associated with a query (lines 11 and 12).

When the iterator object is associated with a query (lines 11 and 12 in Figure 9.9), the program fetches the query result from the database and sets the iterator to a position before the first row in the result of the query. This becomes the current row for the iterator. Subsequently, next operations are issued on the iterator; each moves the iterator to the next row in the result of the query, making it the current row. If the row exists, the

12. We discuss iterators in more detail in Chapter 21 when we discuss object databases.
operation retrieves the attribute values for that row into the corresponding program variables. If no more rows exist, the next operation returns null, and can thus be used to control the looping.

In Figure 9.9, the command (e.next()) in line 13 performs two functions: It gets the next tuple in the query result and controls the while loop. Once we are done with the query result, the command e.close() (line 16) closes the iterator.

Next, consider the same example using positional iterators as shown in Figure 9.10 (program segment J2B). Line 9 in Figure 9.10 shows how a positional iterator type Emppos is declared. The main difference between this and the named iterator is that there are no attribute names in the positional iterator—only attribute types. They still must be compatible with the attribute types in the SQL query result and in the same order. Line 10 shows how a positional iterator variable e of type Emppos is created in the program and then associated with a query (lines 11 and 12).

The positional iterator behaves in a manner that is more similar to embedded SQL (see Section 9.4.2). A `fetch <iterator variable> into <program variables>` command is needed to get the next tuple in a query result. The first time `fetch` is executed, it gets the first tuple (line 13 in Figure 9.10). Line 16 gets the next tuple until no more tuples exist in the query result. To control the loop, a positional iterator function `e.endFetch()` is used. This function is set to a value of `TRUE` when the iterator is initially associated with an SQL query (line 11), and is set to `FALSE` each time a fetch command returns a valid tuple from the query result. It is set to `TRUE` again when a fetch command does not find any more tuples. Line 14 shows how the looping is controlled by negation.
Embedded SQL (see Section 9.4) is sometimes referred to as a static database programming approach because the query text is written within the program and cannot be changed without recompiling or reprocessing the source code. The use of function calls is a more dynamic approach for database programming than embedded SQL. We already saw one dynamic database programming technique—dynamic SQL—in Section 9.4.3. The techniques discussed here provide another approach to dynamic database programming. A library of functions, also known as an application programming interface (API), is used to access the database. Although this provides more flexibility because no preprocessor is needed, one drawback is that syntax and other checks on SQL commands have to be done at runtime. Another drawback is that it sometimes requires more complex programming to access query results because the types and numbers of attributes in a query result may not be known in advance.

In this section, we give an overview of two function call interfaces. We first discuss SQL/CLI (Call Level Interface), which is part of the SQL standard. This was developed as a follow-up to the earlier technique known as ODBC (Open Data Base Connectivity). We use C as the host language in our SQL/CLI examples. Then we give an overview of JDBC, which is the call function interface for accessing databases from JAVA. Although it is commonly assumed that JDBC stands for Java Data Base Connectivity, JDBC is just a registered trademark of Sun Microsystems, not an acronym.

FIGURE 9.10 Program segment J2B, a JAVA program segment that uses a positional iterator to print employee information in a particular department
The main advantage of using a function call interface is that it makes it easier to access multiple databases within the same application program, even if they are stored under different DBMS packages. We discuss this further in Section 9.5.2 when we discuss JAVA database programming with JDBC, although this advantage also applies to database programming with SQL/CLI and ODBC (see Section 9.5.1).

9.5.1 Database Programming with SQL/CLI
Using C as the Host Language

Before using the function calls in SQL/CLI, it is necessary to install the appropriate library packages on the database server. These packages are obtained from the vendor of the DBMS being used. We now give an overview of how SQL/CLI can be used in a C program. We shall illustrate our presentation with the example program segment CLI1 shown in Figure 9.11.

When using SQL/CLI, the SQL statements are dynamically created and passed as string parameters in the function calls. Hence, it is necessary to keep track of the information about host program interactions with the database in runtime data structures, because the database commands are processed at runtime. The information is kept in four types of

```c
//Program CLI1:
0) #include sqlcli.h ;
1) void printSal() {
2) SQLHSTMT stmt1 ;
3) SQLHDBC con1 ;
4) SQLHENV env1 ;
5) SQLRETURN ret1, ret2, ret3, ret4 ;
6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz", SQL_NTS) else exit ;
9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select LNAME, SALARY from EMPLOYEE where SSN = ?", SQL_NTS) ;
11) prompt("Enter a Social Security Number: ", ssn) ;
12) SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15) SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16) SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17) ret2 = SQLFetch(stmt1) ;
18) if (!ret2) printf(ssn, lname, salary)
19) else printf("Social Security Number does not exist: ", ssn) ;
20) }
21) }
```

**FIGURE 9.11** Program segment CLI1, a C program segment with SQL/CLI
9.5 Database Programming with Function Calls: SQL/CLI and JDBC

records, represented as structs in C data types. An environment record is used as a container to keep track of one or more database connections and to set environment information. A connection record keeps track of the information needed for a particular database connection. A statement record keeps track of the information needed for one SQL statement. A description record keeps track of the information about tuples or parameters—for example, the number of attributes and their types in a tuple, or the number and types of parameters in a function call.

Each record is accessible to the program through a C pointer variable—called a handle to the record. The handle is returned when a record is first created. To create a record and return its handle, the following SQL/CLI function is used:

```c
SQLAllocHandle(<handle_type>, <handle_1>, <handle_2>)
```

In this function, the parameters are as follows:

- `<handle_type>` indicates the type of record being created. The possible values for this parameter are the keywords SQL_HANDLE_ENV, SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC, for an environment, connection, statement, or description record, respectively.
- `<handle_1>` indicates the container within which the new handle is being created. For example, for a connection record this would be the environment within which the connection is being created, and for a statement record this would be the connection for that statement.
- `<handle_2>` is the pointer (handle) to the newly created record of type `<handle_type>`.

When writing a C program that will include database calls through SQL/CLI, the following are the typical steps that are taken. We illustrate the steps by referring to the example CLI1 in Figure 9.11, which reads a social security number of an employee and prints the employee’s last name and salary.

1. The library of functions comprising SQL/CLI must be included in the C program. This is called sqlcli.h, and is included using line 0 in Figure 9.11.
2. Declare handle variables of types SQLHSTMT, SQLHDBC, SQLHENV, and SQLHDESC for the statements, connections, environments, and descriptions needed in the program, respectively (lines 2 to 4). Also declare variables of type SQLRETURN (line 5) to hold the return codes from the SQL/CLI function calls. A return code of 0 (zero) indicates successful execution of the function call.
3. An environment record must be set up in the program using SQLAllocHandle. The function to do this is shown in line 6. Because an environment record is not contained in any other record, the parameter `<handle_1>` is the null handle SQL_NULL_HANDLE (null pointer) when creating an environment. The handle (pointer) to the newly created environment record is returned in variable `env1` in line 6.

13. We will not show description records here, to keep our presentation simple.
4. A connection record is set up in the program using SQLAllocHandle. In line 7, the connection record created has the handle con1 and is contained in the environment env1. A connection is then established in con1 to a particular server database using the SQLConnect function of SQL/CLI (line 8). In our example, the database server name we are connecting to is “dbs”, and the account name and password for login are “js” and “xyz”, respectively.

5. A statement record is set up in the program using SQLAllocHandle. In line 9, the statement record created has the handle stmt1 and uses the connection con1.

6. The statement is prepared using the SQL/CLI function SQLPrepare. In line 10, this assigns the SQL statement string (the query in our example) to the statement handle stmt1. The question mark (?) symbol in line 10 represents a statement parameter, which is a value to be determined at runtime—typically by binding it to a C program variable. In general, there could be several parameters. They are distinguished by the order of appearance of the question marks in the statement (the first ? represents parameter 1, the second ? represents parameter 2, and so on). The last parameter in SQLPrepare should give the length of the SQL statement string in bytes, but if we enter the keyword SQL_NTS, this indicates that the string holding the query is a null-terminated string so that SQL can calculate the string length automatically. This also applies to other string parameters in the function calls.

7. Before executing the query, any parameters should be bound to program variables using the SQL/CLI function SQLBindParameter. In Figure 9.11, the parameter (indicated by ?) to the prepared query referenced by stmt1 is bound to the C program variable ssn in line 12. If there are n parameters in the SQL statement, we should have n SQLBindParameter function calls, each with a different parameter position (1, 2, ..., n).

8. Following these preparations, we can now execute the SQL statement referenced by the handle stmt1 using the function SQLExecute (line 13). Notice that although the query will be executed in line 13, the query results have not yet been assigned to any C program variables.

9. In order to determine where the result of the query is returned, one common technique is the bound columns approach. Here, each column in a query result is bound to a C program variable using the SQLBindCol function. The columns are distinguished by their order of appearance in the SQL query. In Figure 9.11 lines 15 and 16, the two columns in the query (LNAME and SALARY) are bound to the C program variables lname and salary, respectively.\(^\text{14}\)

10. Finally, in order to retrieve the column values into the C program variables, the function SQLFetch is used (line 17). This function is similar to the FETCH command of embedded SQL. If a query result has a collection of tuples, each

\(^{14}\) An alternative technique known as unbound columns uses different SQL/CLI functions, namely SQLGetCol or SQLGetData, to retrieve columns from the query result without previously binding them; these are applied after the SQLFetch command in step 17.
SQLFetch call gets the next tuple and returns its column values into the bound program variables. SQLFetch returns an exception (nonzero) code if there are no more tuples.\textsuperscript{15}

As we can see, using dynamic function calls requires a lot of preparation to set up the SQL statements and to bind parameters and query results to the appropriate program variables.

In CLI\textsuperscript{1} a single tuple is selected by the SQL query. Figure 9.12 shows an example of retrieving multiple tuples. We assume that appropriate C program variables have been declared as in Figure 9.2. The program segment in CLI\textsuperscript{2} reads (inputs) a department number and then retrieves the employees who work in that department. A loop then iterates over each employee record, one at a time, and prints the employee's last name and salary.

\texttt{//Program Segment CLI2:}
0) #include sqlcli.h ;
1) void printDepartmentEmps() {
2) SQLHSTMT stmt1 ;
3) SQLHDBC con1 ;
4) SQLENV env1 ;
5) SQLRETURN ret1, ret2, ret3, ret4 ;
6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz", SQL_NTS) else exit ;
9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select LNAME, SALARY from EMPLOYEE where DNO = ?", SQL_NTS) ;
11) prompt("Enter the Department Number: ", dno) ;
12) SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15) SQLBindCol(stmt1, 1, SQL_CHAR, & lname, 15, &fetchlen1) ;
16) SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17) ret2 = SQLFetch(stmt1) ;
18) while (!ret2) {
19) printf(lname, salary) ;
20) ret2 = SQLFetch(stmt1) ;
21) }
22) }
23) }

**FIGURE 9.12** Program segment CLI2, a C program segment that uses SQL/CLI for a query with a collection of tuples in its result.

\textsuperscript{15} If unbound program variables are used, SQLFetch returns the tuple into a temporary program area. Each subsequent SQLGetCol (or SQLGetData) returns one attribute value in order.
9.5.2 JDBC: SQL Function Calls for JAVA Programming

We now turn our attention to how SQL can be called from the JAVA object-oriented programming language. The function libraries for this access are known as JDBC. The JAVA programming language was designed to be platform independent—that is, a program should be able to run on any type of computer system that has a JAVA interpreter installed. Because of this portability, many RDBMS vendors provide JDBC drivers so that it is possible to access their systems via JAVA programs. A JDBC driver is basically an implementation of the function calls specified in the JDBC API (Application Programming Interface) for a particular vendor’s RDBMS. Hence, a JAVA program with JDBC function calls can access any RDBMS that has a JDBC driver available.

Because JAVA is object-oriented, its function libraries are implemented as classes. Before being able to process JDBC function calls with JAVA, it is necessary to import the JDBC class libraries, which are called java.sql.*. These can be downloaded and installed via the Web.

JDBC is designed to allow a single JAVA program to connect to several different databases. These are sometimes called the data sources accessed by the JAVA program. These data sources could be stored using RDBMSs from different vendors and could reside on different machines. Hence, different data source accesses within the same JAVA program may require JDBC drivers from different vendors. To achieve this flexibility, a special JDBC class called the driver manager class is employed, which keeps track of the installed drivers. A driver should be registered with the driver manager before it is used. The operations (methods) of the driver manager class include getDriver, registerDriver, and deregisterDriver. These can be used to add and remove drivers dynamically. Other functions set up and close connections to data sources, as we shall see.

To load a JDBC driver explicitly, the generic JAVA function for loading a class can be used. For example, to load the JDBC driver for the ORACLE RDBMS, the following command can be used:

```java
Class.forName("oracle.jdbc.driver.OracleDriver")
```

This will register the driver with the driver manager and make it available to the program. It is also possible to load and register the driver(s) needed in the command line that runs the program, for example, by including the following in the command line:

```
-Djdbc.drivers = oracle.jdbc.driver
```

The following are typical steps that are taken when writing a JAVA application program with database access through JDBC function calls. We illustrate the steps by referring to the example JDBC1 in Figure 9.13, which reads a social security number of an employee and prints the employee’s last name and salary.

---

16. This section assumes familiarity with object-oriented concepts and basic JAVA concepts. If readers lack this familiarity, they should postpone this section until after reading Chapter 20.

17. As we mentioned earlier, JDBC is a registered trademark of Sun Microsystems, although it is commonly thought to be an acronym for Java Data Base Connectivity.

18. These are available from several Web sites—for example, through the Web site at the URL http://industry.java.sun.com/products/jdbc/drivers.
1. The JDBC library of classes must be imported into the JAVA program. These classes are called java.sql.*, and can be imported using line 1 in Figure 9.13. Any additional JAVA class libraries needed by the program must also be imported.

2. Load the JDBC driver as discussed previously (lines 4 to 7). The JAVA exception in line 5 occurs if the driver is not loaded successfully.

3. Create appropriate variables as needed in the JAVA program (lines 8 and 9).

4. A connection object is created using the getConnection function of the DriverManager class of JDBC. In lines 12 and 13, the connection object is created by using the function call getConnection(urlstring), where urlstring has the form

   jdbc:oracle:<driverType>:<dbaccount>/<password>

   An alternative form is

   getConnection(url, dbaccount, password)

   Various properties can be set for a connection object, but they are mainly related to transactional properties, which we discuss in Chapter 17.

5. A statement object is created in the program. In JDBC, there is a basic statement class, Statement, with two specialized subclasses: PreparedStatement and CallableStatement. This example illustrates how PreparedStatement objects are created and used. The next example (Figure 9.14) illustrates the other type of Statement objects. In line 14, a query string with a single parameter—indicated by the “?” symbol—is created in the variable stmt1. In line 15, an object p of type PreparedStatement is created based on the query string in stmt1 and using the connection object conn. In general, the programmer should use PreparedStatement objects if a query is to be executed multiple times, since it would be prepared, checked, and compiled only once, thus saving this cost for the additional executions of the query.

6. The question mark (?) symbol in line 14 represents a statement parameter, which is a value to be determined at runtime, typically by binding it to a JAVA program variable. In general, there could be several parameters, distinguished by the order of appearance of the question marks (first ? represents parameter 1, second ? represents parameter 2, and so on) in the statement, as discussed previously.

7. Before executing a PreparedStatement query, any parameters should be bound to program variables. Depending on the type of the parameter, functions such as setString, setInteger, setDouble, and so on are applied to the PreparedStatement object to set its parameters. In Figure 9.13, the parameter (indicated by ?) in object p is bound to the JAVA program variable ssn in line 18. If there are n parameters in the SQL statement, we should have n Set… functions, each with a different parameter position (1, 2, …, n). Generally, it is advisable to clear all parameters before setting any new values (line 17).

8. Following these preparations, we can now execute the SQL statement referenced by the object p using the function executeQuery (line 19). There is a generic function execute in JDBC, plus two specialized functions: executeUpdate and
executeQuery. executeUpdate is used for SQL insert, delete, or update statements, and returns an integer value indicating the number of tuples that were affected. executeQuery is used for SQL retrieval statements, and returns an object of type ResultSet, which we discuss next.

9. In line 19, the result of the query is returned in an object r of type ResultSet. This resembles a two-dimensional array or a table, where the tuples are the rows and the attributes returned are the columns. A ResultSet object is similar to a cursor in embedded SQL and an iterator in SQLJ. In our example, when the query is executed, r refers to a tuple before the first tuple in the query result. The r.next() function (line 20) moves to the next tuple (row) in the ResultSet object and returns null if there are no more objects. This is used to control the looping. The programmer can refer to the attributes in the current tuple using various get... functions that depend on the type of each attribute (for example, getString, getInteger, getDouble, and so on). The programmer can either use the attribute positions (1, 2) or the actual attribute names ("LNAME", "SALARY")
with the `get_` functions. In our examples, we used the positional notation in lines 21 and 22.

In general, the programmer can check for SQL exceptions after each JDBC function call. Notice that JDBC does not distinguish between queries that return single tuples and those that return multiple tuples, unlike some of the other techniques. This is justifiable because a single tuple result set is just a special case.

In example JDBC1, a single tuple is selected by the SQL query, so the loop in lines 20 to 24 is executed at most once. The next example, shown in Figure 9.14, illustrates the retrieval of multiple tuples. The program segment in JDBC2 reads (inputs) a department number and then retrieves the employees who work in that department. A loop then iterates over each employee record, one at a time, and prints the employee's last name and salary. This example also illustrates how we can execute a query directly, without having to prepare it as in the previous example. This technique is preferred for queries

```java
//Program Segment JDBC2:
0) import java.io.*;
1) import java.sql.*;
2) class printDepartmentEmps {
3)   public static void main (String args []) throws SQLException, IOException {
4)     try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)       } catch (ClassNotFoundException x) {
6)         System.out.println ("Driver could not be loaded");
7)       }
8)     String dbacct, passwd, lname ;
9)     Double salary ;
10)    Integer dno ;
11)    dbacct = readentry("Enter database account:" );
12)    passwd = readentry("Enter password:" );
13)    Connection conn = DriverManager.getConnection
14)       ("jdbc:oracle:oci8:" + dbacct + "/" + passwd);
15)    dno = readentry("Enter a Department Number:" );
16)    String q = "select LNAME, SALARY from EMPLOYEE where DNO = " +
17)       dno.toString() ;
18)    Statement s = conn.createStatement() ;
19)    ResultSet r = s.executeQuery(q) ;
20)   while (r.next()) {
21)       lname = r.getString(1) ;
22)       salary = r.getDouble(2) ;
23)       System.out.println(lname + salary) ;
24)   }
```

**FIGURE 9.14** Program segment JDBC2, a Java program segment that uses JDBC for a query with a collection of tuples in its result
that will be executed only once, since it is simpler to program. In line 17 of Figure 9.14, the programmer creates a `Statement` object (instead of `PreparedStatement`, as in the previous example) without associating it with a particular query string. The query string `q` is passed to the statement object `s` when it is executed in line 18.

This concludes our brief introduction to JDBC. The interested reader is referred to the Web site http://java.sun.com/docs/books/tutorial/jdbc/, which contains many further details on JDBC.

### 9.6 DATABASE STORED PROCEDURES AND SQL/PSM

We conclude this chapter with two additional topics related to database programming. In Section 9.6.1, we discuss the concept of stored procedures, which are program modules that are stored by the DBMS at the database server. Then in Section 9.6.2, we discuss the extensions to SQL that are specified in the standard to include general-purpose programming constructs in SQL. These extensions are known as SQL/PSM (SQL/Persistent Stored Modules) and can be used to write stored procedures. SQL/PSM also serves as an example of a database programming language that extends a database model and language—namely, SQL—with some programming constructs, such as conditional statements and loops.

#### 9.6.1 Database Stored Procedures and Functions

In our presentation of database programming techniques so far, there was an implicit assumption that the database application program was running on a client machine that is different from the machine on which the database server—and the main part of the DBMS software package—is located. Although this is suitable for many applications, it is sometimes useful to create database program modules—procedures or functions—that are stored and executed by the DBMS at the database server. These are historically known as database **stored procedures**, although they can be functions or procedures. The term used in the SQL standard for stored procedures is **persistent stored modules**, because these programs are stored persistently by the DBMS, similarly to the persistent data stored by the DBMS.

Stored procedures are useful in the following circumstances:

- If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.
- Executing a program at the server can reduce data transfer and hence communication cost between the client and server in certain situations.
- These procedures can enhance the modeling power provided by views by allowing more complex types of derived data to be made available to the database users. In addition, they can be used to check for complex constraints that are beyond the specification power of assertions and triggers.

In general, many commercial DBMSs allow stored procedures and functions to be written in a general-purpose programming language. Alternatively, a stored procedure can...
be made of simple SQL commands such as retrievals and updates. The general form of
declaring a stored procedures is as follows:

```
CREATE PROCEDURE <procedure name> ( <parameters> )
<local declarations>
<procedure body> ;
```

The parameters and local declarations are optional, and are specified only if needed.
For declaring a function, a return type is necessary, so the declaration form is

```
CREATE FUNCTION <function name> ( <parameters> )
RETURNS <return type>
<local declarations>
<function body> ;
```

If the procedure (or function) is written in a general-purpose programming language,
it is typical to specify the language, as well as a file name where the program code is
stored. For example, the following format can be used:

```
CREATE PROCEDURE <procedure name> ( <parameters> )
LANGUAGE <programming language name>
EXTERNAL NAME <file path name> ;
```

In general, each parameter should have a parameter type that is one of the SQL data
types. Each parameter should also have a parameter mode, which is one of IN, OUT, or
INOUT. These correspond to parameters whose values are input only, output (returned)
only, or both input and output, respectively.

Because the procedures and functions are stored persistently by the DBMS, it should
be possible to call them from the various SQL interfaces and programming techniques.
The CALL statement in the SQL standard can be used to invoke a stored procedure—
either from an interactive interface or from embedded SQL or SQLJ. The format of the
statement is as follows:

```
CALL <procedure or function name> ( <argument list> ) ;
```

If this statement is called from JDBC, it should be assigned to a statement object of type
CallableStatement (see Section 9.5.2).

### 9.6.2 SQL/PSM: Extending SQL for Specifying Persistent Stored Modules

SQL/PSM is the part of the SQL standard that specifies how to write persistent stored modules.
It includes the statements to create functions and procedures that we described in the previ-
ous section. It also includes additional programming constructs to enhance the power of SQL
for the purpose of writing the code (or body) of stored procedures and functions.

In this section, we discuss the SQL/PSM constructs for conditional (branching)
statements and for looping statements. These will give a flavor of the type of constructs
that SQL/PSM has incorporated. Then we give an example to illustrate how these constructs can be used.

The conditional branching statement in SQL/PSM has the following form:

\[
\text{IF } \text{condition} \text{ THEN } \text{statement list} \\
\quad \text{ELSEIF } \text{condition} \text{ THEN } \text{statement list} \\
\quad \ldots \\
\quad \text{ELSEIF } \text{condition} \text{ THEN } \text{statement list} \\
\quad \text{ELSE } \text{statement list} \\
\quad \text{END IF ;}
\]

Consider the example in Figure 9.15, which illustrates how the conditional branch structure can be used in an SQL/PSM function. The function returns a string value (line 1) describing the size of a department based on the number of employees. There is one IN integer parameter, \texttt{deptno}, which gives a department number. A local variable \texttt{NoOfEmps} is declared in line 2. The query in lines 3 and 4 returns the number of employees in the department, and the conditional branch in lines 5 to 8 then returns one of the values \{"HUGE", "LARGE", "MEDIUM", "SMALL"\} based on the number of employees.

SQL/PSM has several constructs for looping. There are standard while and repeat looping structures, which have the following forms:

\[
\text{WHILE } \text{condition} \text{ DO} \\
\quad \text{statement list} \\
\text{END WHILE ;}
\]

19. We only give a brief introduction to SQL/PSM here. There are many other features in the SQL/PSM standard.
There is also a cursor-based looping structure. The statement list in such a loop is executed once for each tuple in the query result. This has the following form:

```
FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
    <statement list>
END FOR;
```

Loops can have names, and there is a `LEAVE <loop name>` statement to break a loop when a condition is satisfied. SQL/PSM has many other features, but they are outside the scope of our presentation.

**Summary**

In this chapter we presented additional features of the SQL database language. In particular, we presented an overview of the most important techniques for database programming. We started in Section 9.1 by presenting the features for specifying general constraints as assertions. Then we discussed the concept of a view in SQL. We then discussed the various approaches to database application programming in Sections 9.3 to 9.6.

**Review Questions**

9.1. How does SQL allow implementation of general integrity constraints?
9.2. What is a view in SQL, and how is it defined? Discuss the problems that may arise when one attempts to update a view. How are views typically implemented?
9.3. List the three main approaches to database programming. What are the advantages and disadvantages of each approach?
9.4. What is the impedance mismatch problem? Which of the three programming approaches minimizes this problem?
9.5. Describe the concept of a cursor and how it is used in embedded SQL.
9.6. What is SQLJ used for? Describe the two types of iterators available in SQLJ.

**Exercises**

9.7. Consider the database shown in Figure 1.2, whose schema is shown in Figure 2.1. Write a program segment to read a student's name and print his or her grade point average, assuming that A=4, B=3, C=2, and D=1 points. Use embedded SQL with C as the host language.
9.8. Repeat Exercise 9.7, but use SQLJ with JAVA as the host language.
9.9. Consider the `LIBRARY` relational database schema of Figure 6.12. Write a program segment that retrieves the list of books that became overdue yesterday and that
prints the book title and borrower name for each. Use embedded SQL with C as
the host language.

9.10. Repeat Exercise 9.9, but use SQL with JAVA as the host language.

9.11. Repeat Exercises 9.7 and 9.9, but use SQL/CLI with C as the host language.

9.12. Repeat Exercises 9.7 and 9.9, but use JDBC with JAVA as the host language.

9.13. Repeat Exercise 9.7, but write a function in SQL/PSM.

9.14. Specify the following views in SQL on the COMPANY database schema shown in
Figure 5.5.

a. A view that has the department name, manager name, and manager salary for
every department.

b. A view that has the employee name, supervisor name, and employee salary for
each employee who works in the ‘Research’ department.

c. A view that has the project name, controlling department name, number of
employees, and total hours worked per week on the project for each project.

d. A view that has the project name, controlling department name, number of
employees, and total hours worked per week on the project for each project
with more than one employee working on it.

9.15. Consider the following view, DEPT_SUMMARY, defined on the COMPANY database of Fig-
ure 5.6:

```
CREATE VIEW DEPT_SUMMARY (D, C, TOTAL_S, AVERAGE_S)
AS
SELECT DNO,
COUNT (*), SUM (SALARY), AVG (SALARY)
FROM EMPLOYEE
GROUP BY DNO;
```

State which of the following queries and updates would be allowed on the view.
If a query or update would be allowed, show what the corresponding query or
update on the base relations would look like, and give its result when applied to
the database of Figure 5.6.

a. SELECT *
FROM DEPT_SUMMARY;

b. SELECT D, C
FROM DEPT_SUMMARY
WHERE TOTAL_S > 100000;

c. SELECT D, AVERAGE_S
FROM DEPT_SUMMARY
WHERE C > (SELECT C FROM DEPT_SUMMARY WHERE D=4);

d. UPDATE DEPT_SUMMARY
SET D=3
WHERE D=4;

e. DELETE FROM DEPT_SUMMARY
WHERE C > 4;
Selected Bibliography

The question of view updates is addressed by Dayal and Bernstein (1978), Keller (1982), and Langerak (1990), among others. View implementation is discussed in Blakeley et al. (1989). Negri et al. (1991) describes formal semantics of sql queries.