FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

part 3
The Relational Data Model and SQL

chapter 7

More SQL: Complex Queries, Triggers, Views, and Schema Modification

This chapter describes more advanced features of the SQL language for relational databases. We start in Section 7.1 by presenting more complex features of SQL retrieval queries, such as nested queries, joined tables, outer joins, aggregate functions, and grouping, and case statements. In Section 7.2, we describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. We also introduce the concept of triggers and the CREATE TRIGGER statement, which will be presented in more detail in Section 26.1 when we present the principles of active databases. Then, in Section 7.3, we describe the SQL facility 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. Section 7.4 introduces the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints. Section 7.5 is the chapter summary.

This chapter is a continuation of Chapter 6. The instructor may skip parts of this chapter if a less detailed introduction to SQL is intended.

7.1 More Complex SQL Retrieval Queries

In Section 6.3, we described some basic types of retrieval queries in SQL. Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex retrievals from the database. We discuss several of these features in this section.

7.1.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. Recall from Section 5.1.2 that NULL is used to represent a missing value, but that it usually has one of three different interpretations—value *unknown* (value exists but is not known, or it is not known whether or not the value exists), value *not available* (value exists but is purposely withheld), or value *not applicable* (the attribute does not apply to this tuple or is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL.

1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database. An example of the other case of unknown would be NULL for a person's home phone because it is not known whether or not the person has a home phone.
2. **Unavailable or withheld value.** A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. **Not applicable attribute.** An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

It is often not possible to determine which of the meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish among the different meanings of NULL.

In general, each individual NULL value is considered to be different from every other NULL value in the various database records. When a record with NULL in one of its attributes is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used. Table 7.1 shows the resulting values.

Table 7.1 Logical Connectives in Three-Valued Logic

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

In Tables 7.1(a) and 7.1(b), the rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query. Each expression result would have a value of TRUE, FALSE, or UNKNOWN. The result of combining the two values using the AND logical connective is shown by the entries in Table 7.1(a). Table 7.1(b) shows the result of using the OR logical connective. For example, the result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN. Table 7.1(c) shows the result of the NOT logical operation. Notice that in standard Boolean logic, only TRUE or FALSE values are permitted; there is no UNKNOWN value.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected. However, there are exceptions to that rule for certain operations, such as outer joins, as we shall see in Section 7.1.6.

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **IS** or IS **NOT**. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN; see Section 7.1.6). Query 18 illustrates NULL comparison by retrieving any employees who do not have a supervisor.

Query 18. Retrieve the names of all employees who do not have supervisors.

```
Q18:    SELECT    Fname, Lname
        FROM      EMPLOYEE
        WHERE     Super_ssn IS NULL;
```

7.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within another SQL query. That other query is called the **outer query**. These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed. Query 4 is formulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A. Q4A introduces the comparison operator **IN**, which compares a value v with a set (or multiset) of values V and evaluates to **TRUE** if v is one of the elements in V.

In Q4A, the first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, whereas the second nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as worker. In the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

```
Q4A:    SELECT    DISTINCT Pnumber
        FROM      PROJECT
        WHERE     Pnumber IN
                  ( SELECT    Pnumber
                    FROM      PROJECT, DEPARTMENT, EMPLOYEE
                    WHERE     Dnum = Dnumber AND
                              Mgr_ssn = Ssn AND Lname = 'Smith' )
                  OR
                  Pnumber IN
                  ( SELECT    Pno
                    FROM      WORKS_ON, EMPLOYEE
                    WHERE     Essn = Ssn AND Lname = 'Smith' );
```

If a nested query returns a single attribute *and* a single tuple, the query result will be a single (**scalar**) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a **table** (relation), which is a set or multiset of tuples.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT    DISTINCT Essn
FROM      WORKS_ON
WHERE     (Pno, Hours) IN    ( SELECT    Pno, Hours
                               FROM      WORKS_ON
                               WHERE     Essn = '123456789' );
```

This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on. In this example, the IN operator compares the subtuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset v (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value v is equal to *some value* in the set V and is hence equivalent to IN. The two keywords ANY and SOME have the same effect. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than *all* the values in the set (or multiset) V. An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT    Lname, Fname
FROM      EMPLOYEE
WHERE     Salary > ALL    ( SELECT    Salary
                            FROM      EMPLOYEE
                            WHERE     Dno = 5 );
```

Notice that this query can also be specified using the MAX aggregate function (see Section 7.1.7).

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the *outer query*, and another in a relation in the FROM clause of the *nested query*. The rule is that a reference to an *unqualified attribute* refers to the relation declared in the **innermost nested query**. For example, in the SELECT clause and WHERE clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, we specify and refer to an *alias* (tuple variable) for that relation. These rules are similar to scope rules for program variables in most programming languages that allow nested procedures and functions. To illustrate the potential ambiguity of attribute names in nested queries, consider Query 16.

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:  SELECT   E.Fname, E.Lname
      FROM     EMPLOYEE AS E
      WHERE    E.Ssn IN ( SELECT   D.Essn
                          FROM     DEPENDENT AS D
                          WHERE    E.Fname = D.Dependent_name
                                   AND E.Sex = D.Sex );
```

In the nested query of Q16, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex. If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT. However, we would not *have to* qualify the attributes Fname and Ssn of EMPLOYEE if they appeared in the nested query because the DEPENDENT relation does not have attributes called Fname and Ssn, so there is no ambiguity.

It is generally advisable to create tuple variables (aliases) for *all the tables referenced in an SQL query* to avoid potential errors and ambiguities, as illustrated in Q16.

7.1.3 Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated**. We can understand a correlated query better by considering that the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query.* For example, we can think of Q16 as follows: For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, Q16 may be written as in Q16A:

```
Q16A:   SELECT   E.Fname, E.Lname
        FROM     EMPLOYEE AS E, DEPENDENT AS D
        WHERE    E.Ssn = D.Essn AND E.Sex = D.Sex
                 AND E.Fname = D.Dependent_name;
```

7.1.4 The EXISTS and UNIQUE Functions in SQL

EXISTS and UNIQUE are Boolean functions that return TRUE or FALSE; hence, they can be used in a WHERE clause condition. The EXISTS function in SQL is used to check whether the result of a nested query is *empty* (contains no tuples) or not. The result of EXISTS is a Boolean value **TRUE** if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples. We illustrate the use of EXISTS—and NOT EXISTS—with some examples. First, we formulate Query 16 in an alternative form that uses EXISTS as in Q16B:

```
Q16B:   SELECT   E.Fname, E.Lname
        FROM     EMPLOYEE AS E
        WHERE    EXISTS ( SELECT   *
                          FROM     DEPENDENT AS D
                          WHERE    E.Ssn = D.Essn AND E.Sex = D.Sex
                                   AND E.Fname = D.Dependent_name);
```

EXISTS and NOT EXISTS are typically used in conjunction with a *correlated* nested query. In Q16B, the nested query references the Ssn, Fname, and Sex attributes of the EMPLOYEE relation from the outer query. We can think of Q16B as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Essn, Sex, and Dependent_name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and returns **FALSE** otherwise. On the other hand, NOT EXISTS(Q) returns **TRUE** if there are *no tuples* in the result of nested query Q, and returns **FALSE** otherwise. Next, we illustrate the use of NOT EXISTS.

Query 6. Retrieve the names of employees who have no dependents.

```
Q6:     SELECT   Fname, Lname
        FROM     EMPLOYEE
        WHERE    NOT EXISTS ( SELECT   *
                              FROM     DEPENDENT
                              WHERE    Ssn = Essn );
```

In Q6, the correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If *none exist,* the EMPLOYEE tuple is selected because the **WHERE**-clause condition will evaluate to **TRUE** in this case. We can explain Q6 as follows: For *each* EMPLOYEE tuple, the correlated nested query selects all

DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

Query 7. List the names of managers who have at least one dependent.

```
Q7:   SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    EXISTS ( SELECT   *
                        FROM     DEPENDENT
                        WHERE    Ssn = Essn )
               AND
               EXISTS ( SELECT   *
                        FROM     DEPARTMENT
                        WHERE    Ssn = Mgr_ssn );
```

One way to write this query is shown in Q7, where we specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, we select the EMPLOYEE tuple. Can you rewrite this query using only a single nested query or no nested queries?

The query Q3: *Retrieve the name of each employee who works on* all *the projects controlled by department number 5* can be written using EXISTS and NOT EXISTS in SQL systems. We show two ways of specifying this query Q3 in SQL as Q3A and Q3B. This is an example of certain types of queries that require *universal quantification*, as we will discuss in Section 8.6.7. One way to write this query is to use the construct (*S2* EXCEPT *S1*) as explained next, and checking whether the result is empty.[1] This option is shown as Q3A.

```
Q3A:  SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    NOT EXISTS ( ( SELECT   Pnumber
                              FROM     PROJECT
                              WHERE    Dnum = 5)
                              EXCEPT   ( SELECT   Pno
                                         FROM     WORKS_ON
                                         WHERE    Ssn = Essn) );
```

In Q3A, the first subquery (which is not correlated with the outer query) selects all projects controlled by department 5, and the second subquery (which is correlated) selects all projects that the particular employee being considered works on. If the set difference of the first subquery result MINUS (EXCEPT) the second subquery result is empty, it means that the employee works on all the projects and is therefore selected.

[1]Recall that EXCEPT is the set difference operator. The keyword MINUS is also sometimes used, for example, in Oracle.

The second option is shown as Q3B. Notice that we need two-level nesting in Q3B and that this formulation is quite a bit more complex than Q3A.

```
Q3B:  SELECT   Lname, Fname
      FROM     EMPLOYEE
      WHERE    NOT EXISTS  ( SELECT   *
                             FROM     WORKS_ON B
                             WHERE    ( B.Pno IN  ( SELECT   Pnumber
                                                    FROM     PROJECT
                                                    WHERE    Dnum = 5 )
                             AND
                             NOT EXISTS  ( SELECT   *
                                           FROM     WORKS_ON C
                                           WHERE    C.Essn = Ssn
                                           AND      C.Pno = B.Pno )));
```

In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose Pno is of a project controlled by department 5, *if* there is not a WORKS_ON (C) tuple with the same Pno and the same Ssn as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on. It corresponds to the way we will write this query in tuple relation calculus (see Section 8.6.7).

There is another SQL function, UNIQUE(Q), which returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set (no duplicates) or a multiset (duplicates exist).

7.1.5 Explicit Sets and Renaming in SQL

We have seen several queries with a nested query in the WHERE clause. It is also possible to use an **explicit set of values** in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

Query 17. Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17:  SELECT   DISTINCT Essn
      FROM     WORKS_ON
      WHERE    Pno IN (1, 2, 3);
```

In SQL, it is possible to **rename** any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names in general, and it can be used in appropriate parts of a query. For example, Q8A shows how query Q8 from Section 4.3.2 can be slightly changed to retrieve the last name of each employee and his or her supervisor while renaming the resulting attribute names

as Employee_name and Supervisor_name. The new names will appear as column headers for the query result.

```
Q8A:  SELECT  E.Lname AS Employee_name, S.Lname AS Supervisor_name
      FROM    EMPLOYEE AS E, EMPLOYEE AS S
      WHERE   E.Super_ssn = S.Ssn;
```

7.1.6 Joined Tables in SQL and Outer Joins

The concept of a **joined table** (or **joined relation**) was incorporated into SQL to permit users to specify a table resulting from a join operation *in the* FROM *clause* of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause. For example, consider query Q1, which retrieves the name and address of every employee who works for the 'Research' department. It may be easier to specify the join of the EMPLOYEE and DEPARTMENT relations in the WHERE clause, and then to select the desired tuples and attributes. This can be written in SQL as in Q1A:

```
Q1A:  SELECT  Fname, Lname, Address
      FROM    (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)
      WHERE   Dname = 'Research';
```

The FROM clause in Q1A contains a single *joined table*. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a **NATURAL JOIN** on two relations *R* and *S*, no join condition is specified; an implicit *EQUIJOIN condition* for *each pair of attributes with the same name* from *R* and *S* is created. Each such pair of attributes is included *only once* in the resulting relation (see Sections 8.3.2 and 8.4.4 for more details on the various types of join operations in relational algebra).

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause. This is illustrated in Q1B, where the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as Dname, Dno (to match the name of the desired join attribute Dno in the EMPLOYEE table), Mssn, and Msdate. The implied join condition for this NATURAL JOIN is EMPLOYEE.Dno = DEPT.Dno, because this is the only pair of attributes with the same name after renaming:

```
Q1B:  SELECT  Fname, Lname, Address
      FROM    (EMPLOYEE NATURAL JOIN
              (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))
      WHERE   Dname = 'Research';
```

The default type of join in a joined table is called an **inner join**, where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in query Q8A, only employees who *have a supervisor* are included in the result;

an EMPLOYEE tuple whose value for Super_ssn is NULL is excluded. If the user requires that all employees be included, a different type of join called **OUTER JOIN** must be used explicitly (see Section 8.4.4 for the definition of OUTER JOIN in relational algebra). There are several variations of OUTER JOIN, as we shall see. In the SQL standard, this is handled by explicitly specifying the keyword OUTER JOIN in a joined table, as illustrated in Q8B:

```
Q8B:  SELECT  E.Lname AS Employee_name,
              S.Lname AS Supervisor_name
      FROM    (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S
              ON E.Super_ssn = S.Ssn);
```

In SQL, the options available for specifying joined tables include INNER JOIN (only pairs of tuples that match the join condition are retrieved, same as JOIN), LEFT OUTER JOIN (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table), RIGHT OUTER JOIN (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table), and FULL OUTER JOIN. In the latter three options, the keyword OUTER may be omitted. If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN). The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation (see Section 8.2.2), although this should be used only with the utmost care because it generates all possible tuple combinations.

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**. For example, Q2A is a different way of specifying query Q2 from Section 6.3.1 using the concept of a joined table:

```
Q2A:  SELECT  Pnumber, Dnum, Lname, Address, Bdate
      FROM    ((PROJECT JOIN DEPARTMENT ON Dnum = Dnumber)
              JOIN EMPLOYEE ON Mgr_ssn = Ssn)
      WHERE   Plocation = 'Stafford';
```

Not all SQL implementations have implemented the new syntax of joined tables. In some systems, a different syntax was used to specify outer joins by using the comparison operators + =, = +, and + = + for left, right, and full outer join, respectively, when specifying the join condition. For example, this syntax is available in Oracle. To specify the left outer join in Q8B using this syntax, we could write the query Q8C as follows:

```
Q8C:  SELECT  E.Lname, S.Lname
      FROM    EMPLOYEE E, EMPLOYEE S
      WHERE   E.Super_ssn + = S.Ssn;
```

7.1.7 Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. **Grouping** is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database

applications, and we will introduce their use in SQL through examples. A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.[2] The COUNT function returns the *number of tuples or values* as specified in a query. The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.[3] We illustrate the use of these functions with several queries.

Query 19. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q19:    SELECT   SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
        FROM     EMPLOYEE;
```

This query returns a *single-row* summary of all the rows in the EMPLOYEE table. We could use AS to rename the column names in the resulting single-row table; for example, as in Q19A.

```
Q19A:   SELECT   SUM (Salary) AS Total_Sal, MAX (Salary) AS Highest_Sal,
                 MIN (Salary) AS Lowest_Sal, AVG (Salary) AS Average_Sal
        FROM     EMPLOYEE;
```

If we want to get the preceding aggregate function values for employees of a specific department—say, the 'Research' department—we can write Query 20, where the EMPLOYEE tuples are restricted by the WHERE clause to those employees who work for the 'Research' department.

Query 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:    SELECT   SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
        FROM     (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber)
        WHERE    Dname = 'Research';
```

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21:    SELECT   COUNT (*)
        FROM     EMPLOYEE;

Q22:    SELECT   COUNT (*)
        FROM     EMPLOYEE, DEPARTMENT
        WHERE    DNO = DNUMBER AND DNAME = 'Research';
```

[2]Additional aggregate functions for more advanced statistical calculation were added in SQL-99.

[3]Total order means that for any two values in the domain, it can be determined that one appears before the other in the defined order; for example, DATE, TIME, and TIMESTAMP domains have total orderings on their values, as do alphabetic strings.

Here the asterisk (*) refers to the *rows* (tuples), so COUNT (*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples, as in the next example.

Query 23. Count the number of distinct salary values in the database.

```
Q23:  SELECT   COUNT (DISTINCT Salary)
      FROM     EMPLOYEE;
```

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute); the only exception is for COUNT(*) because tuples instead of values are counted. In the previous examples, any Salary values that are NULL are not included in the aggregate function calculation. The general rule is as follows: when an aggregate function is applied to a collection of values, NULLs are removed from the collection before the calculation; if the collection becomes empty because all values are NULL, the aggregate function will return NULL (except in the case of COUNT, where it will return 0 for an empty collection of values).

The preceding examples summarize *a whole relation* (Q19, Q21, Q23) or a selected subset of tuples (Q20, Q22), and hence all produce a table with a single row or a single value. They illustrate how functions are applied to retrieve a summary value or summary tuple from a table. These functions can also be used in selection conditions involving nested queries. We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query. For example, to retrieve the names of all employees who have two or more dependents (Query 5), we can write the following:

```
Q5:   SELECT   Lname, Fname
      FROM     EMPLOYEE
      WHERE    ( SELECT   COUNT (*)
                 FROM     DEPENDENT
                 WHERE    Ssn = Essn ) >= 2;
```

The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to two, the employee tuple is selected.

SQL also has aggregate functions SOME and ALL that can be applied to a collection of Boolean values; SOME returns TRUE if at least one element in the collection is TRUE, whereas ALL returns TRUE if all elements in the collection are TRUE.

7.1.8 Grouping: The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions *to subgroups of tuples in a relation,* where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees *in each department* or the number

of employees who work *on each project*. In these cases we need to **partition** the relation into nonoverlapping subsets (or **groups**) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s)**. We can then apply the function to each such group independently to produce summary information about each group. SQL has a **GROUP BY** clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should *also appear in the SELECT clause,* so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

Query 24. For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q24:   SELECT    Dno, COUNT (*), AVG (Salary)
       FROM      EMPLOYEE
       GROUP BY  Dno;
```

In Q24, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the GROUP BY attribute Dno. Hence, each group contains the employees who work in the same department. The COUNT and AVG functions are applied to each such group of tuples. Notice that the SELECT clause includes only the grouping attribute and the aggregate functions to be applied on each group of tuples. Figure 7.1(a) illustrates how grouping works and shows the result of Q24.

If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a *NULL value in the grouping attribute.* For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result of Q24.

Query 25. For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q25:   SELECT    Pnumber, Pname, COUNT (*)
       FROM      PROJECT, WORKS_ON
       WHERE     Pnumber = Pno
       GROUP BY  Pnumber, Pname;
```

Q25 shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied *after* the joining of the two relations in the WHERE clause.

Sometimes we want to retrieve the values of these functions only for *groups that satisfy certain conditions.* For example, suppose that we want to modify Query 25 so that only projects with more than two employees appear in the result. SQL provides a **HAVING** clause, which can appear in conjunction with a GROUP BY clause, for this purpose. HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated by Query 26.

Figure 7.1
Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

(a)

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Borg	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

(b)

Pname	Pnumber	...	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

Query 26. For each project *on which more than two employees work,* retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26:    SELECT     Pnumber, Pname, COUNT (*)
        FROM       PROJECT, WORKS_ON
        WHERE      Pnumber = Pno
        GROUP BY   Pnumber, Pname
        HAVING     COUNT (*) > 2;
```

Notice that although selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups.* Figure 7.1(b) illustrates the use of HAVING and displays the result of Q26.

Query 27. For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
Q27:    SELECT     Pnumber, Pname, COUNT (*)
        FROM       PROJECT, WORKS_ON, EMPLOYEE
        WHERE      Pnumber = Pno AND Ssn = Essn AND Dno = 5
        GROUP BY   Pnumber, Pname;
```

In Q27, we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE clause—namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the aggregate function in the SELECT clause and another to the function in the HAVING clause). For example, suppose that we want to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. Suppose that we write the following *incorrect* query:

```
SELECT     Dno, COUNT (*)
FROM       EMPLOYEE
WHERE      Salary>40000
GROUP BY   Dno
HAVING     COUNT (*) > 5;
```

This is incorrect because it will select only departments that have more than five employees *who each earn more than $40,000.* The rule is that the WHERE clause is executed first, to select individual tuples or joined tuples; the HAVING clause is applied later, to select individual groups of tuples. In the incorrect query, the tuples are already restricted to employees who earn more than $40,000 *before* the function in the HAVING clause is applied. One way to write this query correctly is to use a nested query, as shown in Query 28.

Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28:    SELECT      Dno, COUNT (*)
        FROM        EMPLOYEE
        WHERE       Salary>40000 AND Dno IN
                    ( SELECT      Dno
                      FROM        EMPLOYEE
        GROUP BY    Dno
                      HAVING      COUNT (*) > 5)
        GROUP BY    Dno;
```

7.1.9 Other SQL Constructs: WITH and CASE

In this section, we illustrate two additional SQL constructs. The WITH clause allows a user to define a table that will only be used in a particular query; it is somewhat similar to creating a view (see Section 7.3) that will be used only in one query and then dropped. This construct was introduced as a convenience in SQL:99 and may not be available in all SQL based DBMSs. Queries using WITH can generally be written using other SQL constructs. For example, we can rewrite Q28 as Q28′:

```
Q28′:   WITH        BIGDEPTS (Dno) AS
                    ( SELECT      Dno
                      FROM        EMPLOYEE
                      GROUP BY    Dno
                      HAVING      COUNT (*) > 5)
        SELECT      Dno, COUNT (*)
        FROM        EMPLOYEE
        WHERE       Salary>40000 AND Dno IN BIGDEPTS
        GROUP BY    Dno;
```

In Q28′, we defined in the WITH clause a temporary table BIG_DEPTS whose result holds the Dno's of departments with more than five employees, then used this table in the subsequent query. Once this query is executed, the temporary table BIGDEPTS is discarded.

SQL also has a CASE construct, which can be used when a value can be different based on certain conditions. This can be used in any part of an SQL query where a value is expected, including when querying, inserting or updating tuples. We illustrate this with an example. Suppose we want to give employees different raise amounts depending on which department they work for; for example, employees in department 5 get a $2,000 raise, those in department 4 get $1,500 and those in department 1 get $3,000 (see Figure 5.6 for the employee tuples). Then we could re-write the update operation U6 from Section 6.4.3 as U6′:

```
U6′:    UPDATE      EMPLOYEE
        SET         Salary =
        CASE        WHEN      Dno = 5     THEN Salary + 2000
                    WHEN      Dno = 4     THEN Salary + 1500
                    WHEN      Dno = 1     THEN Salary + 3000
                    ELSE      Salary + 0 ;
```

In U6′, the salary raise value is determined through the CASE construct based on the department number for which each employee works. The CASE construct can also be used when inserting tuples that can have different attributes being NULL depending on the type of record being inserted into a table, as when a specialization (see Chapter 4) is mapped into a single table (see Chapter 9) or when a union type is mapped into relations.

7.1.10 Recursive Queries in SQL

In this section, we illustrate how to write a recursive query in SQL. This syntax was added in SQL:99 to allow users the capability to specify a recursive query in a declarative manner. An example of a **recursive relationship** between tuples of the same type is the relationship between an employee and a supervisor. This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation in Figures 5.5 and 5.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor). An example of a recursive operation is to retrieve all supervisees of a supervisory employee *e* at all levels—that is, all employees *e*′ directly supervised by *e*, all employees *e*″ directly supervised by each employee *e*′, all employees *e*‴ directly supervised by each employee *e*″, and so on. In SQL:99, this query can be written as follows:

```
Q29:    WITH RECURSIVE    SUP_EMP (SupSsn, EmpSsn) AS
              ( SELECT    SupervisorSsn, Ssn
                FROM      EMPLOYEE
                UNION
                SELECT    E.Ssn, S.SupSsn
                FROM      EMPLOYEE AS E, SUP_EMP AS S
                WHERE     E.SupervisorSsn = S.EmpSsn)
        SELECT*
        FROM              SUP_EMP;
```

In Q29, we are defining a view SUP_EMP that will hold the result of the recursive query. The view is initially empty. It is first loaded with the first level (supervisor, supervisee) Ssn combinations via the first part (**SELECT** SupervisorSss, Ssn **FROM** EMPLOYEE), which is called the **base query**. This will be combined via UNION with each successive level of supervisees through the second part, where the view contents are joined again with the base values to get the second level combinations, which are UNIONed with the first level. This is repeated with successive levels until a **fixed point** is reached, where no more tuples are added to the view. At this point, the result of the recursive query is in the view SUP_EMP.

7.1.11 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two—SELECT and FROM—are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are

specified in the following order, with the clauses between square brackets […] being optional:

SELECT <attribute and function list>
FROM <table list>
[**WHERE** <condition>]
[**GROUP BY** <grouping attribute(s)>]
[**HAVING** <group condition>]
[**ORDER BY** <attribute list>];

The SELECT clause lists the attributes or functions to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. GROUP BY specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

In order to formulate queries correctly, it is useful to consider the steps that define the *meaning* or *semantics* of each query. A query is evaluated *conceptually*[4] by first applying the FROM clause (to identify all tables involved in the query or to materialize any joined tables), followed by the WHERE clause to select and join tuples, and then by GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result. If none of the last three clauses (GROUP BY, HAVING, and ORDER BY) are specified, we can *think conceptually* of a query as being executed as follows: For *each combination of tuples*—one from each of the relations specified in the FROM clause—evaluate the WHERE clause; if it evaluates to TRUE, place the values of the attributes specified in the SELECT clause from this tuple combination in the result of the query. Of course, this is not an efficient way to implement the query in a real system, and each DBMS has special query optimization routines to decide on an execution plan that is efficient to execute. We discuss query processing and optimization in Chapters 18 and 19.

In general, there are numerous ways to specify the same query in SQL. This flexibility in specifying queries has advantages and disadvantages. The main advantage is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the IN comparison operator. Some users may be more comfortable with one approach, whereas others may be more comfortable with another. From the programmer's and the system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible.

The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify

[4]The actual order of query evaluation is implementation dependent; this is just a way to conceptually view a query in order to correctly formulate it.

particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way. Ideally, this should not be the case: The DBMS should process the same query in the same way regardless of how the query is specified. But this is quite difficult in practice, since each DBMS has different methods for processing queries specified in different ways. Thus, an additional burden on the user is to determine which of the alternative specifications is the most efficient to execute. Ideally, the user should worry only about specifying the query correctly, whereas the DBMS would determine how to execute the query efficiently. In practice, however, it helps if the user is aware of which types of constructs in a query are more expensive to process than others.

7.2 Specifying Constraints as Assertions and Actions as Triggers

In this section, we introduce two additional features of SQL: the **CREATE ASSERTION** statement and the **CREATE TRIGGER** statement. Section 7.2.1 discusses CREATE ASSERTION, which can be used to specify additional types of constraints that are outside the scope of the *built-in relational model constraints* (primary and unique keys, entity integrity, and referential integrity) that we presented in Section 5.2. These built-in constraints can be specified within the **CREATE TABLE** statement of SQL (see Sections 6.1 and 6.2).

In Section 7.2.2 we introduce **CREATE TRIGGER**, which can be used to specify automatic actions that the database system will perform when certain events and conditions occur. This type of functionality is generally referred to as **active databases**. We only introduce the basics of **triggers** in this chapter, and present a more complete discussion of active databases in Section 26.1.

7.2.1 Specifying General Constraints as Assertions in SQL

In SQL, users can specify general constraints—those that do not fall into any of the categories described in Sections 6.1 and 6.2— via **declarative assertions**, using the **CREATE ASSERTION** statement. 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.Mgr_ssn = 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 disable 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 is **violated**. 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 so that the condition will always be TRUE. Thus, the assertion is violated if the result of the query is not empty. In the preceding 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 *individual* attributes and domains (see Section 6.2.1) and on *individual* tuples (see Section 6.2.4). A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated* in a specific table. 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 simple checks are implemented more efficiently by the DBMS.

7.2.2 Introduction to Triggers in SQL

Another important statement in SQL is CREATE TRIGGER. 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. 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. We discuss triggers in detail in Section 26.1 when we describe *active databases*. Here we just give a simple example of how triggers may be used.

Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database (see Figures 5.5 and 5.6). Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION,[5] which will notify the supervisor. The trigger could then be written as in R5 below. Here we are using the syntax of the Oracle database system.

```
R5: CREATE TRIGGER SALARY_VIOLATION
    BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
      ON EMPLOYEE
    FOR EACH ROW
      WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
                            WHERE SSN = NEW.SUPERVISOR_SSN ) )
                            INFORM_SUPERVISOR(NEW.Supervisor_ssn,
                            NEW.Ssn );
```

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:

1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.
3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically. A complete discussion is given in Section 26.1.

[5]Assuming that an appropriate external procedure has been declared. We discuss stored procedures in Chapter 10.

7.3 Views (Virtual Tables) in SQL

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

7.3.1 Concept of a View in SQL

A **view** in SQL terminology is a single table that is derived from other tables.[6] These other tables can be *base tables* or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically 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, referring to the COMPANY database 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 three tables EMPLOYEE, WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can 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.

7.3.2 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 new 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 7.2 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;
```

[6] 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.

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

Figure 7.2
Two views specified on the database schema of Figure 5.5.

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 the 'ProductX' project, we can utilize the WORKS_ON1 view and specify the query as in QV1:

```
QV1:    SELECT    Fname, Lname
        FROM      WORKS_ON1
        WHERE     Pname = 'ProductX';
```

The same query would require the specification of two joins if specified on the base relations directly; 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 Section 7.3.4 and Chapter 30).

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 does not have to be realized or materialized at the time of *view definition* but rather at the time when 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 kept up-to-date. We will discuss various ways the DBMS can utilize to keep a view up-to-date in the next subsection.

If we do not need a view anymore, 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:    DROP VIEW    WORKS_ON1;
```

7.3.3 View Implementation, View Update, and Inline Views

The problem of how a DBMS can efficiently implement a view for efficient querying is complex. Two main approaches have been suggested. One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the

user) into a query on the underlying base tables. For example, the query QV1 would be automatically modified to the following query by the DBMS:

```
SELECT   Fname, Lname
FROM     EMPLOYEE, PROJECT, WORKS_ON
WHERE    Ssn = Essn AND Pno = Pnumber
         AND Pname = 'ProductX';
```

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 view queries are going to be applied to the same view within a short period of time. The second strategy, called **view materialization**, involves physically creating a temporary or permanent view table when the view is first queried or created 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 the DBMS can determine what new tuples must be inserted, deleted, or modified in a *materialized view table* when a database update is applied *to one of the defining base tables*. The view is generally kept as a materialized (physically stored) table 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 table and recompute it from scratch when future queries reference the view.

Different strategies as to when a materialized view is updated are possible. The **immediate update** strategy updates a view as soon as the base tables are changed; the **lazy update** strategy updates the view when needed by a view query; and the **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).

A user can always issue a retrieval query against any view. However, issuing an INSERT, DELETE, or UPDATE command on a view table is in many cases not possible. 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*. Hence, it is often not possible for the DBMS to determine which of the updates is intended. To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

```
UV1:   UPDATE WORKS_ON1
       SET      Pname = 'ProductY'
       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. In addition, some of these updates will create

additional side effects that affect the result of other queries. For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

```
(a):  UPDATE WORKS_ON
      SET    Pno =    ( SELECT Pnumber
                      FROM   PROJECT
                      WHERE  Pname = 'ProductY' )
      WHERE  Essn IN  ( SELECT Ssn
                      FROM   EMPLOYEE
                      WHERE  Lname = 'Smith' AND Fname = 'John' )
                AND
             Pno =    ( SELECT Pnumber
                      FROM   PROJECT
                      WHERE  Pname = 'ProductX' );

(b):  UPDATE PROJECT   SET    Pname = 'ProductY'
      WHERE  Pname = 'ProductX';
```

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' 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 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 Pname = 'ProductX'.

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

```
UV2:  UPDATE   DEPT_INFO
      SET      Total_sal = 100000
      WHERE    Dname = 'Research';
```

Generally, a view update is feasible when only *one possible update* on the base relations can accomplish the desired update operation on the view. Whenever an update on the view can be mapped to *more than one update* on the underlying base relations, it is usually not permitted. Some researchers have suggested that the DBMS have a certain procedure for choosing one of the possible updates as the most likely one. Some researchers have developed methods for choosing the most likely update, whereas other researchers prefer to have the user choose the desired update mapping during view definition. But these options are generally not available in most commercial DBMSs.

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 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 **WITH CHECK OPTION** should be added at the end of the view definition if a view *is to be updated* by INSERT, DELETE, or UPDATE statements. This allows the system to reject operations that violate the SQL rules for view updates. The full set of SQL rules for when a view may be modified by the user are more complex than the rules stated earlier.

It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view**. In this case, the view is defined within the query itself.

7.3.4 Views as Authorization Mechanisms

We describe SQL query authorization statements (GRANT and REVOKE) in detail in Chapter 30, when we present database security and authorization mechanisms. Here, we will just give a couple of simple examples to illustrate how views can be used to hide certain attributes or tuples from unauthorized users. Suppose a certain user is only allowed to see employee information for employees who work for department 5; then we can create the following view DEPT5EMP and grant the user the privilege to query the view but not the base table EMPLOYEE itself. This user will only be able to retrieve employee information for employee tuples whose Dno = 5, and will not be able to see other employee tuples when the view is queried.

```
CREATE VIEW   DEPT5EMP   AS
SELECT        *
FROM          EMPLOYEE
WHERE         Dno = 5;
```

In a similar manner, a view can restrict a user to only see certain columns; for example, only the first name, last name, and address of an employee may be visible as follows:

```
CREATE VIEW   BASIC_EMP_DATA   AS
SELECT        Fname, Lname, Address
FROM          EMPLOYEE;
```

Thus by creating an appropriate view and granting certain users access to the view and not the base tables, they would be restricted to retrieving only the data specified in the view. Chapter 30 discusses security and authorization in detail, including the GRANT and REVOKE statements of SQL.

7.4 Schema Change Statements in SQL

In this section, we give an overview of the **schema evolution commands** available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements. This can be done while the database is operational and does not require recompilation of the database schema. Certain

checks must be done by the DBMS to ensure that the changes do not affect the rest of the database and make it inconsistent.

7.4.1 The DROP Command

The DROP command can be used to drop *named* schema elements, such as tables, domains, types, or constraints. One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command. There are two *drop behavior* options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

```
DROP SCHEMA COMPANY CASCADE;
```

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements* in it; otherwise, the DROP command will not be executed. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 6.1, we can get rid of the DEPENDENT relation by issuing the following command:

```
DROP TABLE DEPENDENT CASCADE;
```

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views (see Section 7.3) or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition* from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command (see Section 6.4.2) should be used instead of DROP TABLE.

The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

7.4.2 The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible **alter table actions** include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema (see Figure 6.1), we can use the command

```
ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);
```

We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple (see Section 6.4.3). If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other schema elements) reference the column. For example, the following command removes the attribute Address from the EMPLOYEE base table:

```
ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN Address CASCADE;
```

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

```
ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn
    DROP DEFAULT;
ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn
    SET DEFAULT '333445555';
```

One can also change the constraints specified on a table by adding or dropping a named constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 6.2 from the EMPLOYEE relation, we write:

```
ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK CASCADE;
```

Once this is done, we can redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the **ADD CONSTRAINT** keyword in the ALTER TABLE statement followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed.

The preceding subsections gave an overview of the schema evolution commands of SQL. It is also possible to create new tables and views within a database schema using the appropriate commands. There are many other details and options; we refer the interested reader to the SQL documents listed in the Selected Bibliography at the end of this chapter.

7.5 Summary

In this chapter we presented additional features of the SQL database language. We started in Section 7.1 by presenting more complex features of SQL retrieval queries, including nested queries, joined tables, outer joins, aggregate functions, and grouping. In Section 7.2, we described the CREATE ASSERTION statement, which allows the specification of more general constraints on the database, and introduced the

concept of triggers and the CREATE TRIGGER statement. Then, in Section 7.3, we described the SQL facility 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. Section 7.4 introduced the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints.

Table 7.2 summarizes the syntax (or structure) of various SQL statements. This summary is not meant to be comprehensive or to describe every possible SQL construct; rather, it is meant to serve as a quick reference to the major types of constructs available in SQL. We use BNF notation, where nonterminal symbols

Table 7.2 Summary of SQL Syntax

CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>] { , <column name> <column type> [<attribute constraint>] } [<table constraint> { , <table constraint> }])
DROP TABLE <table name> ALTER TABLE <table name> ADD <column name> <column type>
SELECT [DISTINCT] <attribute list> FROM (<table name> { <alias> } | <joined table>) { , (<table name> { <alias> } | <joined table>) } [WHERE <condition>] [GROUP BY <grouping attributes> [HAVING <group selection condition>]] [ORDER BY <column name> [<order>] { , <column name> [<order>] }]
<attribute list> ::= (* | (<column name> | <function> (([DISTINCT] <column name> | *))) { , (<column name> | <function> (([DISTINCT] <column name> | *)) }))
<grouping attributes> ::= <column name> { , <column name> }
<order> ::= (ASC | DESC)
INSERT INTO <table name> [(<column name> { , <column name> })] (VALUES (<constant value> , { <constant value> }) { , (<constant value> { , <constant value> }) } | <select statement>)
DELETE FROM <table name> [WHERE <selection condition>]
UPDATE <table name> SET <column name> = <value expression> { , <column name> = <value expression> } [WHERE <selection condition>]
CREATE [UNIQUE] INDEX <index name> ON <table name> (<column name> [<order>] { , <column name> [<order>] }) [CLUSTER]
DROP INDEX <index name>
CREATE VIEW <view name> [(<column name> { , <column name> })] AS <select statement>
DROP VIEW <view name>

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

are shown in angled brackets < … >, optional parts are shown in square brackets […], repetitions are shown in braces { … }, and alternatives are shown in parentheses (… | … | …).[7]

Review Questions

7.1. Describe the six clauses in the syntax of an SQL retrieval query. Show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?

7.2. Describe conceptually how an SQL retrieval query will be executed by specifying the conceptual order of executing each of the six clauses.

7.3. Discuss how NULLs are treated in comparison operators in SQL. How are NULLs treated when aggregate functions are applied in an SQL query? How are NULLs treated if they exist in grouping attributes?

7.4. Discuss how each of the following constructs is used in SQL, and discuss the various options for each construct. Specify what each construct is useful for.

a. Nested queries
b. Joined tables and outer joins
c. Aggregate functions and grouping
d. Triggers
e. Assertions and how they differ from triggers
f. The SQL WITH clause
g. SQL CASE construct
h. Views and their updatability
i. Schema change commands

Exercises

7.5. Specify the following queries on the database in Figure 5.5 in SQL. Show the query results if each query is applied to the database state in Figure 5.6.

a. For each department whose average employee salary is more than $30,000, retrieve the department name and the number of employees working for that department.
b. Suppose that we want the number of *male* employees in each department making more than $30,000, rather than all employees (as in Exercise 7.5a). Can we specify this query in SQL? Why or why not?

[7]The full syntax of SQL is described in many voluminous documents of hundreds of pages.

7.6. Specify the following queries in SQL on the database schema in Figure 1.2.

a. Retrieve the names and major departments of all straight-A students (students who have a grade of A in all their courses).

b. Retrieve the names and major departments of all students who do not have a grade of A in any of their courses.

7.7. In SQL, specify the following queries on the database in Figure 5.5 using the concept of nested queries and other concepts described in this chapter.

a. Retrieve the names of all employees who work in the department that has the employee with the highest salary among all employees.

b. Retrieve the names of all employees whose supervisor's supervisor has '888665555' for Ssn.

c. Retrieve the names of employees who make at least $10,000 more than the employee who is paid the least in the company.

7.8. 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*

7.9. Consider the following view, DEPT_SUMMARY, defined on the COMPANY database in Figure 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 in 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

Reisner (1977) describes a human factors evaluation of SEQUEL, a precursor of SQL, in which she found that users have some difficulty with specifying join conditions and grouping correctly. Date (1984) contains a critique of the SQL language that points out its strengths and shortcomings. Date and Darwen (1993) describes SQL2. ANSI (1986) outlines the original SQL standard. Various vendor manuals describe the characteristics of SQL as implemented on DB2, SQL/DS, Oracle, INGRES, Informix, and other commercial DBMS products. Melton and Simon (1993) give a comprehensive treatment of the ANSI 1992 standard called SQL2. Horowitz (1992) discusses some of the problems related to referential integrity and propagation of updates in SQL2.

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.

There are many books that describe various aspects of SQL. For example, two references that describe SQL-99 are Melton and Simon (2002) and Melton (2003). Further SQL standards—SQL 2006 and SQL 2008—are described in a variety of technical reports; but no standard references exist.

chapter 8

The Relational Algebra and Relational Calculus

In this chapter we discuss the two *formal languages* for the relational model: the relational algebra and the relational calculus. In contrast, Chapters 6 and 7 described the *practical language* for the relational model, namely the SQL standard. Historically, the relational algebra and calculus were developed before the SQL language. SQL is primarily based on concepts from relational calculus and has been extended to incorporate some concepts from relational algebra as well. Because most relational DBMSs use SQL as their language, we presented the SQL language first.

Recall from Chapter 2 that a data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining the database's structure and constraints. We presented the structures and constraints of the formal relational model in Chapter 5. The basic set of operations for the formal relational model is the **relational algebra**. These operations enable a user to specify basic retrieval requests as *relational algebra expressions*. The result of a retrieval query is a new relation. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

The relational algebra is very important for several reasons. First, it provides a formal foundation for relational model operations. Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs), as we shall discuss in Chapters 18 and 19. Third, some of its concepts are incorporated into the SQL standard

PART II

APPLICATION DEVELOPMENT

6

DATABASE APPLICATION DEVELOPMENT

- How do application programs connect to a DBMS?
- How can applications manipulate data retrieved from a DBMS?
- How can applications modify data in a DBMS?
- What are cursors?
- What is JDBC and how is it used?
- What is SQLJ and how is it used?
- What are stored procedures?
- **Key concepts:** Embedded SQL, Dynamic SQL, cursors; JDBC, connections, drivers, ResultSets, java.sql, SQLJ; stored procedures, SQL/PSM

He profits most who serves best.

------Ivlotto for Rotary International

In Chapter 5, we looked at a wide range of SQL query constructs, treating SQL as an independent language in its own right. A relational DBMS supports an *interactive SQL* interface, and users can directly enter SQL commands. This simple approach is fine as long as the task at hand can be accomplished entirely with SQL cormnands. In practice, we often encounter situations in which we need the greater flexibility of a general-purpose programming language in addition to the data manipulation facilities provided by SQL. For example, we rnay want to integrate a database application with a nice graphical user interface, or we may want to integrate with other existing applications.

Applications that rely on the DBMS to manage data run as separate processes that connect to the DBlvIS to interact with it. Once a connection is established, SQL commands can be used to insert, delete, and modify data. SQL queries can be used to retrieve desired data. but we need to bridge an important difference in how a database system sees data and how an application program in a language like Java or C sees data: The result of a database query is a set (or multiset) or records, hut Java has no set or multiset data type. This mismatch is resolved through additional SQL constructs that allow applications to obtain a handle on a collection and iterate over the records one at a time.

We introduce Embedded SQL, Dynamic SQL, and cursors in Section 6.1. Embedded SQL allows us to access data using static SQL queries in application code (Section 6.1.1); with Dynamic SQL, we can create the queries at run-time (Section 6.1.3). Cursors bridge the gap between set-valued query answers and programming languages that do not support set-values (Section 6.1.2).

The emergence of Java as a popular application development language, especially for Internet applications, has made accessing a DBMS from Java code a particularly important topic. Section 6.2 covers JDBC, a prograruming interface that allows us to execute SQL queries from a Java program and use the results in the Java program. JDBC provides greater portability than Embedded SQL or Dynamic SQL, and offers the ability to connect to several DBMSs without recompiling the code. Section 6.4 covers SQLJ, which does the same for static SQL queries, but is easier to program in than Java, with JDBC.

Often, it is useful to execute application code at the database server, rather than just retrieve data and execute application logic in a separate process. Section 6.5 covers stored procedures, which enable application logic to be stored and executed at the database server. We conclude the chapter by discussing our B&N case study in Section 6.6.

While writing database applications, we must also keep in mind that typically many application programs run concurrently. The transaction concept, introduced in Chapter 1, is used to encapsulate the effects of an application on the database. An application can select certain transaction properties through SQL cormnands to control the degree to which it is exposed to the changes of other concurrently running applications. We touch on the transaction concept at many points in this chapter, and, in particular, cover transaction-related aspects of JDBC. A full discussion of transaction properties and SQL's support for transactions is deferred until Chapter 16.

Exainples that appear in this chapter are available online at

http://www.cs.wisc.edu/-dbbook

6.1 ACCESSING DATABASES FROM APPLICATIONS

In this section, we cover how SQL commands can be executed from within a program in a host language such as C or Java. The use of SQL commands within a host language program is called Embedded SQL. Details of Embedded SQL also depend on the host language. Although similar capabilities are supported for a variety of host languages, the syntax sometimes varies.

We first cover the basics of Embedded SQL with static SQL queries in Section 6.1.1. We then introduce cursors in Section 6.1.2. We discuss Dynamic SQL, which allows us to construct SQL queries at runtime (and execute them) in Section 6.1.3.

6.1.1 Embedded SQL

Conceptually, embedding SQL commands in a host language program is straightforward. SQL statements (i.e., not declarations) can be used wherever a statement in the host language is allowed (with a few restrictions). SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Also, any host language variables used to pass arguments into an SQL command must be declared in SQL. In particular, some special host language variables *must* be declared in SQL (so that, for example, any error conditions arising during SQL execution can be communicated back to the main application program in the host language).

There are, however, two complications to bear in mind. First, the data types recognized by SQL may not be recognized by the host language and vice versa. This mismatch is typically addressed by casting data values appropriately before passing them to or frorn SQL commands. (SQL, like other programming languages, provides an operator to cast values of alle type into values of another type.) The second complication has to do with SQL being set-oriented, and is addressed using cursors (see Section 6.1.2. Commands operate on and produce tables, which are sets

In our discussion of Embedded SQL, we assume that the host language is C for concreteness, because minor differcnces exist in how SQL statements are embedded in differcnt host languages.

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host-language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC

SQL END DECLARE SECTION. The declarations are similar to how they would look in a C program and, as usual in C. are separated by semicolons. For example. we can declare variables *c_sname*, *c_sid*, *c_rating*, and *c_age* (with the initial *c* used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

The first question that arises is which SQL types correspond to the various C types, since we have just declared a collection of C variables whose values are intended to be read (and possibly set) in an SQL run-time environment when an SQL statement that refers to them is executed. The SQL-92 standard defines such a correspondence between the host language types and SQL types for a number of host languages. In our example, *c_snamc* has the type CHARACTER(20) when referred to in an SQL statement, *c_sid* has the type INTEGER, *c_rating* has the type SMALLINT, and *c_age* has the type REAL.

We also need some way for SQL to report what went wrong if an error condition arises when executing an SQL statement. The SQL-92 standard recognizes two special variables for reporting errors, SQLCODE and SQLSTATE. SQLCODE is the older of the two and is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes. SQLSTATE, introduced in the SQL-92 standard for the first time, associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported. One of these two variables *must* be declared. The appropriate C type for SQLCODE is long and the appropriate C type for SQLSTATE is char [6J , that is, a character string five characters long. (Recall the null-terminator in C strings.) In this chapter, we assume that SQLSTATE is declared.

Embedding SQL Statements

All SQL staternents embedded within a host program must be clearly marked, with the details dependent on the host language; in C, SQL statements must be prefixed by EXEC SQL. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

As a simple example, the following Embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the Sailors relation:

```
EXEC SQL
INSERT INTO Sailors VALUES (:c_sname, :c_sid, :c_rating, :c_age);
```

Observe that a semicolon terminates the command, as per the convention for terminating statements in C.

The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task:

```
EXEC SQL WHENEVER [SQLERROR | NOT FOUND] [ CONTINUE | GOTO st'mt ]
```

The intent is that the value of SQLSTATE should be checked after each Embedded SQL statement is executed. If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to *stmt,* which is presumably responsible for error and exception handling. Control is also transferred to *stmt* if NOT FOUND is specified and the value of SQLSTATE is 02000, which denotes NO DATA.

6.1.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an *impedance mismatch* occurs because SQL operates on *sets* of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation.

This mechanism is called a cursor. We can declare a cursor on any relation or on any SQL query (because every query returns a set of rows). Once a cursor is declared, we can open it (which positions the cursor just before the first row); fetch the next row; move the cursor (to the next row, to the row after the next *n,* to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage

Cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement:

- \Ve usually need to open a cursor if the embedded statement is a SELECT (i.e.) a query). However, we can avoid opening a cursor if the answer contains a single row, as we see shortly.
- INSERT, DELETE, and UPDATE staternents typically require no cursor, although some variants of DELETE and UPDATE use a cursor.

As an example, we can find the name and age of a sailor, specified by assigning a value to the host variable *c_sid*, declared earlier, as follows:

```
EXEC SQL SELECT S.sname, S.age
         INTO    :c_sname, :c_age
         FROM    Sailors S
         WHERE   S.sid = :c_sid;
```

The INTO clause allows us to assign the columns of the single answer row to the host variables *c_sname* and *c_age*. Therefore, we do not need a cursor to embed this query in a host language program. But what about the following query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable *c_minrating*?

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating
```

This query returns a collection of rows, not just one row. 'When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the cOlnmand with EXEC SQL, how can the answers be bound to host language variables? The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating;
```

This code can be included in a C program, and once it is executed, the cursor *sinfo* is defined. Subsequently, we can open the cursor:

```
OPEN sinfo;
```

The value of *c_minrating* in the SQL query associated with the cursor is the value of this variable when we open the cursor. (The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.)

A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row. We can use the FETCH command to read the first row of cursor *sinfo* into host language variables:

```
FETCH sinfo INTO :c_sname, :c_age;
```

When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement (say, in a while-loop in the C program), we can read all the rows computed by the query, one row at a time. Additional parameters to the FETCH command allow us to position a cursor in very flexible ways, but we do not discuss them.

How do we know when we have looked at all the rows associated with the cursor? By looking at the special variables SQLCODE or SQLSTATE, of course. SQLSTATE, for example, is set to the value 02000, which denotes NO DATA, to indicate that there are no more rows if the FETCH statement positions the cursor after the last row.

When we are done with a cursor, we can close it:

```
CLOSE sinfo;
```

It can be opened again if needed, and the value of : *c_minrating* in the SQL query associated with the cursor would be the value of the host variable *c_minrating* at that time.

Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR
        [WITH HOLD]
        FOR some query
        [ ORDER BY order-item-list ]
        [ FOR READ ONLY | FOR UPDATE ]
```

A cursor can be declared to be a read-only cursor (FOR READ ONLY) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (FOR UPDATE). If it is IIpdatable, simple variants of the UPDATE and

DELETE commands allow us to update or delete the row on which the cursor is positioned. For example, if *sinfa* is an updatable cursor and open, we can execute the following statement:

```
UPDATE  Sailors S
SET     S.rating = S.rating - 1
WHERE   CURRENT of sinfo;
```

This Embedded SQL statement modifies the *rating* value of the row currently pointed to by cursor *sinfa;* similarly, we can delete this row by executing the next statement:

```
DELETE  Sailors S
WHERE   CURRENT of sinfo;
```

A cursor is updatable by default unless it is a scrollable or insensitive cursor (see below), in which case it is read-only by default.

If the keyword SCROLL is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed.

If the keyword INSENSITIVE is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. Otherwise, and by default, other actions of some transaction could modify these rows, creating unpredictable behavior. For example, while we are fetching rows using the *sinfa* cursor, we might modify *rating* values in Sailor rows by concurrently executing the command:

```
UPDATE  Sailors S
SET     S.rating = S.rating -
```

Consider a Sailor row such that (1) it has not yet been fetched, and (2) its original *rating* value would have met the condition in the WHERE clause of the query associated with *sinfa,* but the new *rating* value does not. Do we fetch such a Sailor row'? If INSENSITIVE is specified, the behavior is as if all answers were computed and stored when *sinfo* was opened; thus, the update command has no effect on the rows fetched by *sinfa* if it is executed after *sinfo* is opened. If INSENSITIVE is not specified, the behavior is implementation dependent in this situation.

A holdable cursor is specified using the WITH HOLD clause, and is not closed when the transaction is conunitted. The motivation for this cornes from long

transactions in which we access (and possibly change) a large number of rows of a table. If the transaction is aborted for any reason, the system potentially has to redo a lot of work when the transaction is restarted. Even if the transaction is not aborted, its locks are held for a long time and reduce the concurrency of the system. The alternative is to break the transaction into several smaller transactions, but remembering our position in the table between transactions (and other similar details) is complicated and error-prone. Allowing the application program to commit the transaction it initiated, while retaining its handle on the active table (i.e., the cursor) solves this problem: The application can commit its transaction and start a new transaction and thereby save the changes it has made thus far.

Finally, in what order do FETCH commands retrieve rows? In general this order is unspecified, but the optional ORDER BY clause can be used to specify a sort order. Note that columns mentioned in the ORDER BY clause cannot be updated through the cursor!

The order-item-list is a list of order-items; an order-item is a column name, optionally followed by one of the keywords ASC or DESC. Every column mentioned in the ORDER BY clause must also appear in the select-list of the query associated with the cursor; otherwise it is not clear what columns we should sort on. The keywords ASC or DESC that follow a column control whether the result should be sorted-with respect to that column-in ascending or descending order; the default is ASC. This clause is applied as the last step in evaluating the query.

Consider the query discussed in Section 5.5.1, and the answer shown in Figure 5.13. Suppose that a cursor is opened on this query, with the clause:

```
ORDER BY minage ASC, rating DESC
```

The answer is sorted first in ascending order by *minage,* and if several rows have the same *minage* value, these rows are sorted further in descending order by *rating.* The cursor would fetch the rows in the order shown in Figure 6.1.

rating	*minage*
8	25.5
3	25.5
7	35.0

Figure 6.1 Order in which Tuples Are Fetched

6.1.3 Dynamic SQL

Consider an application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS. Such an application must accept commands from a user and, based on what the user needs, generate appropriate SQL statements to retrieve the necessary data. In such situations, we may not be able to predict in advance just what SQL statements need to be executed, even though there is (presumably) some algorithm by which the application can construct the necessary SQL statements once a user's command is issued.

SQL provides some facilities to deal with such situations; these are referred to as Dynamic SQL. We illustrate the two main commands, PREPARE and EXECUTE, through a simple example:

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

The first statement declares the C variable *c_sqlstring* and initializes its value to the string representation of an SQL command. The second statement results in this string being parsed and compiled as an SQL command, with the resulting executable bound to the SQL variable *readytogo*. (Since *readytogo* is an SQL variable, just like a cursor name, it is not prefixed by a colon.) The third statement executes the command.

Many situations require the use of Dynamic SQL. However, note that the preparation of a Dynamic SQL command occurs at run-time and is run-time overhead. Interactive and Embedded SQL commands can be prepared once at compile-time and then re-executecl as often as desired. Consequently you should limit the use of Dynamic SQL to situations in which it is essential.

There are many more things to know about Dynamic SQL—how we can pass parameters from the host language program to the SQL statement being pre-parcel, for example--but we do not discuss it further.

6.2 AN INTRODUCTION TO JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. As described in Section 6.1.1, a DBMS-specific preprocessor transforms the Embedded SQL statements into function calls in the host language. The details of this translation vary across DBMSs, and therefore even though the source code can be cOlnpiled to work with different DBMSs, the final executable works only with one specific DBMS.

ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language. Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an application programming interface (API). In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs *without recompilation.* Thus, while Embedded SQL is DBMS-independent only at the source code level, applications using ODBC or JDBC are DBMS-independent at the source code level and at the level of the executable. In addition, using ODBC or JDBC, an application can access not just one DBMS but several different ones simultaneously.

ODBC and JDBC achieve portability at the level of the executable by introducing an extra level of indirection. All direct interaction with a specific DBMS happens through a DBMS-specific driver. A driver is a software program that translates the ODBC or JDBC calls into DBMS-specific calls. Drivers are loaded dynamically on demand since the DBMSs the application is going to access are known only at run-time. Available drivers are registered with a driver manager.

One interesting point to note is that a driver does not necessarily need to interact with a DBMS that understands SQL. It is sufficient that the driver translates the SQL commands from the application into equivalent commands that the DBMS understands. Therefore, in the remainder of this section, we refer to a data storage subsystem with which a driver interacts as a data source.

An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source. There is no limit on the number of open connections, and an application can have several open connections to different data sources. Each connection has transaction semantics; that is, changes from one connection are visible to other connections only after the connection has committed its changes. While a connection is opcn, transactions are executed by submitting SQL statements, retrieving results, processing errors, and finally committing or rolling back. The application disconnects from the data source to terminate the interaction.

In the remainder of this chapter, we concentrate on JDBC.

JDBC Drivers: The most up-to-date source of .IDBC drivers is the Sun JDBC Driver page at
http://industry.java.sun.com/products/jdbc/drivers
JDBC drivers are available for all major database sytems.

6.2.1 Architecture

The architecture of JDBC has four main components: the *application,* the *driver manager,* several data source specific *drivers*, and the corresponding *data SOUTces.*

The *application* initiates and terminates the connection with a data source. It sets transaction boundaries, submits SQL statements, and retrieves the results-----all through a well-defined interface as specified by the JDBC API. The primary goal of the *driver manager* is to load JDBC drivers and pass JDBC function calls from the application to the correct driver. The driver manager also handles JDBC initialization and information calls from the applications and can log all function calls. In addition, the driver manager performs some rudimentary error checking. The *driver* establishes the connection with the data source. In addition to submitting requests and returning request results, the driver translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard. The *data source* processes commands from the driver and returns the results.

Depending on the relative location of the data source and the application, several architectural scenarios are possible. Drivers in JDBC are cla.ssified into four types depending on the architectural relationship between the application and the data source:

- Type I Bridges: This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS. An example is a JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source. The application loads only one driver, the bridge. Bridges have the advantage that it is easy to piggy-back the applica.tion onto an existing installation, and no new drivers have to be installed. But using bridges has several drawbacks. The increased number of layers between data source and application affects performance. In addition, the user is limited to the functionality that the ODBC driver supports.
- Type II Direct Translation to the Native API via Non-Java Driver: This type of driver translates JDBC function calls directly into method invocations of the API of one specific data source. The driver is

usually written using a combination of C++ and Java; it is dynamically linked and specific to the data source. This architecture performs significantly better than a JDBC-ODBC bridge. One disadvantage is that the database driver that implements the API needs to be installed on each computer that runs the application.

- Type III—Network Bridges: The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations. In this case, the driver on the client site (Le., the network bridge) is not DBMS-specific. The JDBC driver loaded by the application can be quite small, as the only functionality it needs to implement is sending of SQL statements to the middleware server. The middleware server can then use a Type II JDBC driver to connect to the data source.
- Type IV-Direct Translation to the Native API via Java Driver: Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets. In this case, the driver on the client side is written in Java, but it is DBMS-specific. It translates JDBC calls into the native API of the database system. This solution does not require an intermediate layer, and since the implementation is all Java, its performance is usually quite good.

6.3 JDBC CLASSES AND INTERFACES

JDBC is a collection of Java classes and interfaces that enables database access from prograrl1s written in the Java language. It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling. The classes and interfaces are part of the `java.sql` package. Thus, all code fragments in the remainder of this section should include the statement `import java.sql.*` at the beginning of the code; we omit this statement in the remainder of this section. JDBC 2.0 also includes the `javax.sql` package, the JDBC Optional Package. The package `javax.sql` adds, among other things, the capability of connection pooling and the `RowSet` interface. We discuss connection pooling in Section 6.3.2, and the `ResultSet` interface in Section 6.3.4.

We now illustrate the individual steps that are required to submit a database query to a data source and to retrieve the results.

6.3.1 JDBC Driver Management

In JDBC, data source drivers are managed by the Drivermanager class, which maintains a list of all currently loaded drivers. The Drivermanager class has

methods registerDriver, deregisterDriver, and getDrivers to enable dynamic addition and deletion of drivers.

The first step in connecting to a data source is to load the corresponding JDBC driver. This is accomplished by using the Java mechanism for dynamically loading classes. The static method forName in the Class class returns the Java class as specified in the argument string and executes its `static` constructor. The static constructor of the dynamically loaded class loads an instance of the Driver class, and this Driver object registers itself with the DriverManager class.

The following Java example code explicitly loads a JDBC driver:

```
Class.forName("oracle/jdbc.driver.OracleDriver");
```

There are two other ways ofregistering a driver. We can include the driver with -Djdbc. drivers=oracle/jdbc. driver at the command line when we start the Java application. Alternatively, we can explicitly instantiate a driver, but this method is used only rarely, as the name of the driver has to be specified in the application code, and thus the application becomes sensitive to changes at the driver level.

After registering the driver, we connect to the data source.

6.3.2 Connections

A session with a data source is started through creation of a Connection object; A connection identifies a logical session with a data source; multiple connections within the same Java program can refer to different data sources or the same data source. Connections are specified through a **JDBC** URL, a URL that uses the jdbc protocol. Such a URL has the form

```
jdbc:<subprotocol>:<otherParameters>
```

The code example shown in Figure 6.2 establishes a connection to an Oracle database assuming that the strings userId and password are set to valid values.

In JDBC, connections can have different properties. For example, a connection can specify the granularity of transactions. If autocommit is set for a connection, then each SQL statement is considered to be its own transaction. If autocommit is off, then a series of statements that compose a transaction can be committed using the commit 0 method of the Connection class, or aborted using the rollbackO method. The Connection class has methods to set the

```
String uri = "jdbc:oracle:www.bookstore.com:3083"
Connection connection;
try {
     Connection connection =
          DriverManager.getConnection(urI,userId,password);
}
catch(SQLException excpt) {
     System.out.println(excpt.getMessageO);
     return;
}
```

Figure 6.2 Establishing a Connection with JDBC

JDBC Connections: Remember to close connections to data sources and return shared connections to the connection pool. Database systems have a limited number of resources available for connections, and orphan connections can often only be detected through time-outs-and while the database system is waiting for the connection to time-out, the resources used by the orphan connection are wasted.

autocommit mode (Connection. setAutoCommit) and to retrieve the current autocommit mode (getAutoCommit). The following methods are part of the Connection interface and permit setting and getting other properties:

- public int getTransactionIsolation() throws SQLExceptionand public void setTransactionIsolation(int 1) throws SQLException. These two functions get and set the current level of isolation for transactions handled in the current connection. All five SQL levels of isolation (see Section 16.6 for a full discussion) are possible, and argument *1* can be set as follows:
  - TRANSACTIONJNONE
  - TRANSACTIONJREAD.UNCOMMITTED
  - TRANSACTIONJREAD.COMMITTED
  - TRANSACTIONJREPEATABLEJREAD
  - TRANSACTION.BERIALIZABLE
- public boolean getReadOnlyO throws SQLException and public void setReadOnly(boolean readOnly) throws SQLException. These two functions allow the user to specify whether the transactions executecl through this connection are rcad only.

- public boolean isClosed() throws SQLException.
  Checks whether the current connection has already been closed.
- setAutoCommit and getAutoCommit.
  We already discussed these two functions.

Establishing a connection to a data source is a costly operation since it involves several steps, such as establishing a network connection to the data source, authentication, and allocation of resources such as memory. In case an application establishes many different connections from different parties (such as a Web server), connections are often **pooled** to avoid this overhead. A **connection pool** is a set of established connections to a data source. Whenever a new connection is needed, one of the connections from the pool is used, instead of creating a new connection to the data source.

Connection pooling can be handled either by specialized code in the application, or the optional javax.sql package, which provides functionality for connection pooling and allows us to set different parameters, such as the capacity of the pool, and shrinkage and growth rates. Most application servers (see Section 7.7.2) implement the javax.sql package or a proprietary variant.

6.3.3 Executing SQL Statements

We now discuss how to create and execute SQL statements using JDBC. In the JDBC code examples in this section, we assume that we have a Connection object named con. JDBC supports three different ways of executing statements: Statement, PreparedStatement, and CallableStatement. The Statement class is the base class for the other two statment classes. It allows us to query the data source with any static or dynamically generated SQL query. We cover the PreparedStatement class here and the CallableStatement class in Section 6.5, when we discuss stored procedures.

The PreparedStatement class dynamicaJly generates precompiled SQL statements that can be used several times; these SQL statements can have parameters, but their structure is fixed when the PreparedStatement object (representing the SQL statement) is created.

Consider the sample code using a PreparedStatment object shown in Figure 6.3. The SQL query specifies the query string, but uses '?' for the values of the parameters, which are set later using methods setString, setFloat, and setInt. The "1' placeholders can be used anywhere in SQL statements where they can be replaced with a value. Examples of places where they can appear include the WHERE clause (e.g., 'WHERE author=?'), or in SQL UPDATE and INSERT statements, as in Figure 6.3. The method setString is one way

```
// initial quantity is always zero
String sql = "INSERT INTO Books VALUES(?, 7, ?, ?, 0, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);

// now instantiate the parameters with values
// assume that isbn, title, etc. are Java variables that
// contain the values to be inserted
pstmt.clearParameters();
pstmt.setString(1, isbn);
pstmt.setString(2, title);
pstmt.setString(3, author);
pstmt.setFloat(5, price);
pstmt.setInt(6, year);

int numRows = pstmt.executeUpdate();
```

Figure 6.3 SQL Update Using a PreparedStatement Object

to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use clearParameters() before setting parameter values in order to remove any old data.

There are different ways of submitting the query string to the data source. In the example, we used the executeUpdate command, which is used if we know that the SQL statement does not return any records (SQL UPDATE, INSERT, ALTER, and DELETE statements). The executeUpdate method returns an integer indicating the number of rows the SQL statement modified; it returns 0 for successful execution without modifying any rows.

The executeQuery method is used if the SQL statement returns data, such as in a regular SELECT query. JDBC has its own cursor mechanism in the form of a ResultSet object, which we discuss next. The execute method is more general than executeQuery and executeUpdate; the references at the end of the chapter provide pointers with more details.

6.3.4 ResultSets

As discussed in the previous section, the statement executeQuery returns a ResultSet object, which is similar to a cursor. ResultSet cursors in JDBC 2.0 are very powerful; they allow forward and reverse scrolling and in-place editing and insertions.

In its most basic form, the ResultSet object allows us to read one row of the output of the query at a time. Initially, the ResultSet is positioned before the first row, and we have to retrieve the first row with an explicit call to the next() method. The next method returns false if there are no more rows in the query answer, and true other\vise. The code fragment shown in Figure 6.4 illustrates the basic usage of a ResultSet object.

```
ResultSet rs=stmt.executeQuery(sqlQuery);
// rs is now a cursor
// first call to rs.nextO moves to the first record
// rs.nextO moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next()) {
    // process the data
}
```

Figure 6.4 Using a ResultSet Object

While next() allows us to retrieve the logically next row in the query answer, we can move about in the query answer in other ways too:

- previous() moves back one row.
- absolute(int num) moves to the row with the specified number.
- relative(int num) moves forward or backward (if num is negative) relative to the current position. relative(-1) has the same effect as previous.
- first() moves to the first row, and last() moves to the last row.

Matching Java and SQL Data Types

In considering the interaction of an application with a data source, the issues we encountered in the context of Embedded SQL (e.g., passing information between the application and the data source through shared variables) arise again. To deal with such issues, JDBC provides special data types and specifies their relationship to corresponding SQL data types. Figure 6.5 shows the accessor methods in a ResultSet object for the most common SQL datatypes. With these accessor methods, we can retrieve values from the current row of the query result referenced by the ResultSet object. There are two forms for each accessor method: One method retrieves values by column index, starting at one, and the other retrieves values by column name. The following example shows how to access fields of the current ResultSet row using accesssor methods.

SQL Type	Java class	ResultSet get method
BIT	Boolean	getBooleanO
CHAR	String	getStringO
VARCHAR	String	getStringO
DOUBLE	Double	getDoubleO
FLOAT	Double	getDoubleO
INTEGER	Integer	getIntO
REAL	Double	getFloatO
DATE	java.sql.Date	getDateO
TIME	java.sql.Time	getTimeO
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

Figure 6.5 Reading SQL Datatypes from a ResultSet Object

```
ResultSet rs=stmt.executeQuery(sqIQuery);
String sqlQueryi
ResultSet rs = stmt.executeQuery(sqIQuery)
while (rs.nextO) {
    isbn = rs.getString(l);
    title = rs.getString("TITLE");
    // process isbn and title
}
```

6.3.5 Exceptions and Warnings

Similar to the SQLSTATE variable, most of the methods in java. sql can throw an exception of the type SQLException if an error occurs. The information includes SQLState, a string that describes the error (e.g., whether the statement contained an SQL syntax error). In addition to the standard getMessage 0 method inherited from Throwable, SQLException has two additional methods that provide further information, and a method to get (or chain) additional exceptions:

- public String getSQLState 0 returns an SQLState identifier based on the SQL:1999 specification, as discussed in Section 6.1.1.
- public int getErrorCode () retrieves a vendor-specific error code.
- public SQLException getNextExceptionO gets the next exception in a chain of exceptions associated with the current SQLException object.

An SQL\¥arning is a subclass of SQLException. Warnings are not as severe as errors and the program can usually proceed without special handling of warnings. \Varnings are not thrown like other exceptions, and they are not caught as

part of the try"-catch block around a java. sql statement. We Heed to specifically test whether warnings exist. Connection, Statement, and ResultSet objects all have a getWarnings 0 method with which we can retrieve SQL warnings if they exist. Duplicate retrieval of warnings can be avoided through clearWarnings O. Statement objects clear warnings automatically on execution of the next statement; ResultSet objects clear warnings every time a new tuple is accessed.

Typical code for obtaining SQLWarnings looks similar to the code shown in Figure 6.6.

```
try {
    stmt = con.createStatement();
    warning = con.getWarnings();
    while( warning != null) {
        // handleSQLWarnings          //code to process warning
        warning = warning.getNextWarningO;      //get next warning
    }
    con.clear\Varnings();

    stmt.executeUpdate( queryString );
    warning = stmt.getWarnings();
    while( warning != null) {
        // handleSQLWarnings          //code to process warning
        warning = warning.getNextWarningO;      //get next warning
    }
} // end try
catch ( SQLException SQLe) {
    // code to handle exception
} // end catch
```

Figure 6.6 Processing JDBC Warnings and Exceptions

6.3.6 Examining Database Metadata

We can use the DatabaseMetaData object to obtain information about the database system itself, as well as information frorn the database catalog. For example, the following code fragment shows how to obtain the name and driver version of the JDBC driver:

```
DatabaseMetaData md = con.getMetaD<Lta():

System.out.println("Driver Information:");
```

```
System.out.println("Name:" + md.getDriverNameO
                + "; version:" + mcl.getDriverVersion());
```

The DatabaseMetaData object has many more methods (in JDBC 2.0, exactly 134); we list some methods here:

- public ResultSet getCatalogs0 throws SqLException. This function returns a ResultSet that can be used to iterate over all the catalog relations. The functions getIndexInfo0 and getTables0 work analogously.
- pUblic int getMaxConnections0 throws SqLException. This function returns the ma.ximum number of connections possible.

We will conclude our discussion of JDBC with an example code fragment that examines all database metadata shown in Figure 6.7.

```
DatabaseMetaData dmd = con.getMetaDataO;
ResultSet tablesRS = dmd.getTables(null,null,null,null);
string tableName;

while(tablesRS.next()) {
    tableNarne = tablesRS.getString("TABLE_NAME");

    // print out the attributes of this table
    System.out.println("The attributes of table"
                + tableName + " are:");
    ResultSet columnsRS = dmd.getColums(null,null,tableName, null);
    while (columnsRS.next()) {
        System.out.print(columnsRS.getString("COLUMN_NAME")
                +" ");
    }

    // print out the primary keys of this table
    System.out.println("The keys of table" + tableName + " are:");
    ResultSet keysRS = dmd.getPrimaryKeys(null,null,tableName);
    while (keysRS.next()) {
        'System.out.print(keysRS.getStringC'COLUMN_NAME") +" ");
    }
}
```

Figure 6.7 Obtaining Infon-nation about a Data Source

6.4 SQLJ

SQLJ (short for 'SQL-Java') was developed by the SQLJ Group, a group of database vendors and Sun. SQLJ was developed to complement the dynamic way of creating queries in JDBC with a static model. It is therefore very close to Embedded SQL. Unlike JDBC, having semi-static SQL queries allows the compiler to perform SQL syntax checks, strong type checks of the compatibility of the host variables with the respective SQL attributes, and consistency of the query with the database schema-tables, attributes, views, and stored procedures—all at compilation time. For example, in both SQLJ and Embedded SQL, variables in the host language always are bound statically to the same arguments, whereas in JDBC, we need separate statements to bind each variable to an argument and to retrieve the result. For example, the following SQLJ statement binds host language variables title, price, and author to the return values of the cursor books.

```
#sql books = {
    SELECT title, price INTO :title, :price
    FROM Books WHERE author = :author
};
```

In JDBC, we can dynamically decide which host language variables will hold the query result. In the following example, we read the title of the book into variable ftitle if the book was written by Feynman, and into variable otitle otherwise:

```
// assume we have a ResultSet cursor rs
author = rs.getString(3);

if (author=="Feynman") {
    ftitle = rs.getString(2):
}
else {
    otitle = rs.getString(2);
}
```

When writing SQLJ applications, we just write regular Java code and embed SQL statements according to a set of rules. SQLJ applications are pre-processed through an SQLJ translation program that replaces the embedded SQLJ code with calls to an SQLJ Java library. The modified program code can then be compiled by any Java compiler. Usually the SQLJ Java library makes calls to a JDBC driver, which handles the connection to the database system.

An important philosophical difference exists between Embedded SQL and SQLJ and JDBC. Since vendors provide their own proprietary versions of SQL, it is advisable to write SQL queries according to the SQL-92 or SQL:1999 standard. However, when using Embedded SQL, it is tempting to use vendor-specific SQL constructs that offer functionality beyond the SQL-92 or SQL:1999 standards. SQLJ and JDBC force adherence to the standards, and the resulting code is much more portable across different database systems.

In the remainder of this section, we give a short introduction to SQLJ.

6.4.1 Writing SQLJ Code

We will introduce SQLJ by means of examples. Let us start with an SQLJ code fragment that selects records from the Books table that match a given author.

```
String title; Float price; String atithor;
#sql iterator Books (String title, Float price);
Books books;

// the application sets the author
// execute the query and open the cursor
#sql books = {
    SELECT title, price INTO :title, :price
    FROM Books WHERE author = :author
};

// retrieve results
while (books.next()) {
    System.out.println(books.titleO + ", " + books.price());
}
books.close();
```

The corresponding JDBC code fragment looks as follows (assuming we also declared price, name, and author:

```
PrcparcdStatcment stmt = connection.prepareStatement(
"SELECT title, price FROM Books WHERE author = ?");

// set the parameter in the query ancl execute it
stmt.setString(1, author);
ResultSet 1'8 = stmt.executeQuery();

// retrieve the results
while (rs.next()) {
```

```
System.out.println(rs.getString(1) + ", " + rs.getFloat(2));
}
```

Comparing the JDBC and SQLJ code, we see that the SQLJ code is much easier to read than the JDBC code. Thus, SQLJ reduces software development and maintenance costs.

Let us consider the individual components of the SQLJ code in more detail. All SQLJ statements have the special prefix #sql. In SQLJ, we retrieve the results of SQL queries with iterator objects, which are basically cursors. An iterator is an instance of an iterator class. Usage of an iterator in SQLJ goes through five steps:

- Declare the Iterator Class: In the preceding code, this happened through the statement
  `#sql iterator Books (String title, Float price);`
  This statement creates a new Java class that we can use to instantiate objects.
- Instantiate an Iterator Object from the New Iterator Class: We instantiated our iterator in the statement `Books books;`.
- Initialize the Iterator Using a SQL Statement: In our example, this happens through the statement `#sql books = ....`
- Iteratively, Read the Rows From the Iterator Object: This step is very similar to reading rows through a ResultSet object in JDBC.
- Close the Iterator Object.

There are two types of iterator classes: named iterators and positional iterators. For named iterators, we specify both the variable type and the name of each column of the iterator. This allows us to retrieve individual columns by name as in our previous example where we could retrieve the title column from the Books table using the expression `books.titIe()`. For positional iterators, we need to specify only the variable type for each column of the iterator. To access the individual columns of the iterator, we use a FETCH ... INTO eonstruct, similar to Embedded SQL. Both iterator types have the same performance; which iterator to use depends on the programmer's taste.

Let us revisit our example. \Ve can make the iterator a positional iterator through the following statement:

```
#sql iterator Books (String, Float);
```

We then retrieve the individual rows from the iterator as follows:

```
while (true) {
    #sql { FETCH :books INTO :title, :price, };
    if (books.endFetch()) {
        break:
    }

    // process the book
}
```

6.5 STORED PROCEDURES

It is often important to execute some parts of the application logic directly in the process space of the database system. Running application logic directly at the database has the advantage that the amount of data that is transferred between the database server and the client issuing the SQL statement can be minimized, while at the same time utilizing the full power of the database server.

When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application. If we use a cursor to remotely access the results of an SQL statement, the DBMS has resources such as locks and memory tied up while the application is processing the records retrieved through the cursor. In contrast, a stored procedure is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server. The results can be packaged into one big result and returned to the application, or the application logic can be performed directly at the server, without having to transmit the results to the client at all.

Stored procedures are also beneficial for software engineering rea,sons. Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easy. In addition, application programmers do not need to know the the database schema if we encapsulate all database access into stored procedures.

Although they,are called stored *procedur'es,* they do not have to be procedures in a programming language sense; they can be functions.

6.5.1 Creating a Simple Stored Procedure

Let us look at the example stored procedure written in SQL shown in Figure 6.8. We see that stored procedures must have a name; this stored procedure

has the name 'ShowNumberOfOrders.' Otherwise, it just contains an SQL statement that is precompiled and stored at the server.

```
CREATE PROCEDURE ShowNumberOfOrders
SELECT C.cid, C.cname, COUNT(*)
       FROM    Customers C, Orders a
       WHERE   C.cid = O.cid
       GROUP BY C.cid, C.cname
```

Figure 6.8 A Stored Procedure in SQL

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT. IN parameters are arguments to' the stored procedure. OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process. INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values. Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR.

Let us look at an example of a stored procedure with arguments. The stored procedure shown in Figure 6.9 has two arguments: book_isbn and addedQty. It updates the available number of copies of a book with the quantity from a new shipment.

```
CREATE PROCEDURE AddInventory (
                IN book_isbn CHAR(lO),
                IN addedQty INTEGER)
UPDATE Books
       SET     qty_in_stock = qtyjn_stock + addedQty
       WHERE   bookjsbn = isbn
```

Figure 6.9 A Stored Procedure with Arguments

Stored procedures do not have to be written in SQL; they can be written in any host language. As an example, the stored procedure shown in Figure 6.10 is a Java function that is dynamically executed by the database server whenever it is called by the dient:

6.5.2 Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

```
CREATE PROCEDURE RallkCustomers(IN number INTEGER)
LANGUAGE Java
EXTERNAL NAME 'file:///c:/storedProcedures/rank.jar'
```

Figure 6.10 A Stored Procedure in Java

```
CALL storedProcedureName(argumentl, argument2, ... , argumentN);
```

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure AddInventory would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char isbn[lO];
long qty;
EXEC SQL END DECLARE SECTION

// set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatment class. CallableStatement is a subclass of PreparedStatement and provides the same functionality. A stored procedure could contain multiple SQL staternents or a series of SQL statements-thus, the result could be many different ResultSet objects. We illustrate the case when the stored procedure result is a single ResultSet.

```
CallableStatement cstmt=
            con.prepareCall("{call ShowNumberOfOrders}");
ResultSet rs = cstmt.executeQueryO
while (rs.next())
```

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
// create the cursor class
#sql !terator CustomerInfo(int cid, String cname, int count);

// create the cursor
```

```
CustomerInfo customerinfo;

// call the stored procedure
#sql customerinfo = {CALL ShowNumberOfOrders};
while (customerinfo.nextO) {
    System.out.println(customerinfo.cid() + "," +
                       customerinfo.count());
}
```

6.5.3 SQLIPSM

All major database systems provide ways for users to write stored procedures in a simple, general purpose language closely aligned with SQL. In this section, we briefly discuss the SQL/PSM standard, which is representative of most vendor-specific languages. In PSM, we define modules, which are collections of stored procedures, temporary relations, and other declarations.

In SQL/PSM, we declare a stored procedure as follows:

```
CREATE PROCEDURE name (parameter1,..., parameterN)
    local variable declarations
    procedure code;
```

We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameterl,..., parameterN)
            RETURNS sqIDataType
    local variable declarations
    function code;
```

Each parameter is a triple consisting of the mode (IN, OUT, or INOUT as discussed in the previous section), the parameter name, and the SQL datatype of the parameter. We can seen very simple SQL/PSM procedures in Section 6.5.1. In this case, the local variable declarations were empty, and the procedure code consisted of an SQL query.

We start out with an example of a SQL/PSM function that illustrates the main SQL/PSM constructs. The function takes as input a customer identified by her *cid* and a year. The function returns the rating of the customer, which is defined as follows: Customers who have bought more than ten books during the year are rated 'two'; customer who have purchased between 5 and 10 books are rated 'one', otherwise the customer is rated 'zero'. The following SQL/PSM code computes the rating for a given customer and year.

```
CREATE PROCEDURE RateCustomer
```

```
                    (IN custId INTEGER, IN year INTEGER)
                    RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numOrders INTEGER;
SET numOrders =
        (SELECT COUNT(*) FROM Orders 0 WHERE O.tid = custId);
IF (numOrders> 10) THEN rating=2;
ELSEIF (numOrders>5) THEN rating=1;
ELSE rating=O;
END IF;
RETURN rating;
```

Let us use this example to give a short overview of some SQL/PSM constructs:

- We can declare local variables using the DECLARE statement. In our example, we declare two local variables: 'rating', and 'numOrders'.
- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.
- SQL/PSM has branches and loops. Branches have the following form:

  ```
  IF (condition) THEN statements;
  ELSEIF statements;

  ELSEIF statements;
  ELSE statements; END IF
  ```

  Loops are of the form

  ```
  LOOP
          staternents:
  END LOOP
  ```

- Queries can be used as part of expressions in branches; queries that return a single value can be assigned to variables as in our example above.
- 'We can use the same cursor statements as in Embedded SQL (OPEN, FETCH, CLOSE), but we do not need the EXEC SQL constructs, and variables do not have to be prefixed by a colon ':'.

We only gave a very short overview of SQL/PSM; the references at the end of the chapter provide more information.

6.6 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes finished logical database design, as discussed in Section 3.8, and now consider the queries that they have to support. They expect that the application logic will be implemented in Java, and so they consider JDBC and SQLJ as possible candidates for interfacing the database system with application code.

Recall that DBDudes settled on the following schema:

Books(*isbn:* CHAR(10), *title:* CHAR(8), *author:* CHAR(80),
qty_in_stock: INTEGER, *price:* REAL, *year_published:* INTEGER)
Customers(*cid:* INTEGER, *cname:* CHAR(80), *address:* CHAR(200))
Orders(*ordernum:* INTEGER, *isbn:* CHAR(lO), *cid:* INTEGER,
cardnum: CHAR(l6), *qty:* INTEGER, *order_date:* DATE, *ship_date:* DATE)

Now, DBDudes considers the types of queries and updates that will arise. They first create a list of tasks that will be performed in the application. Tasks performed by customers include the following.

- Customers search books by author name, title, or ISBN.
- Customers register with the website. Registered customers might want to change their contact information. DBDudes realize that they have to augment the Customers table with additional information to capture login and password information for each customer; we do not discuss this aspect any further.
- Customers check out a final shopping basket to complete a sale.
- Customers add and delete books from a 'shopping basket' at the website.
- Customers check the status of existing orders and look at old orders.

Administrative tasks performed by employees of B&N are listed next.

- Employees look up customer contact information.
- Employees add new books to the inventory.
- Employees fulfill orders, and need to update the shipping date of individual books.
- Employees analyze the data to find profitable customers and customers likely to respond to special marketing campaigns.

Next, DBDudes consider the types of queries that will arise out of these tasks. To support searching for books by name, author, title, or ISBN, DBDudes decide to write a stored procedure as follows:

```
CREATE PROCEDURE SearchByISBN (IN book.isbn CHAR(10))
    SELECT B.title, B.author, B.qty_in_stock, B.price, B.yeaLpublished
    FROM   Books B
    WHERE  B.isbn = book.isbn
```

Placing an order involves inserting one or more records into the Orders table. Since DBDudes has not yet chosen the Java-based technology to program the application logic, they assume for now that the individual books in the order are stored at the application layer in a Java array. To finalize the order, they write the following JDBC code shown in Figure 6.11, which inserts the elements from the array into the Orders table. Note that this code fragment assumes several Java variables have been set beforehand.

```
String sql = "INSERT INTO Orders VALUES(7, 7, 7, 7, 7, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);
con.setAutoCommit(false);

try {
    // orderList is a vector of Order objects
    // ordernum is the current order number
    // dd is the ID of the customer, cardnum is the credit card number
    for (int i=O; iiorderList.lengthO; i++)
        // now instantiate the parameters with values
        Order currentOrder = orderList[i];
        pstmt.clearParameters();
        pstmt.setInt(l, ordernum);
        pstmt.setString(2, Order.getlsbnO);
        pstmt.setInt(3, dd);
        pstmt.setString(4, creditCardNum);
        pstmt.setlnt(5, Order.getQtyO);
        pstmt.setDate(6, null);

        pstmt.executeUpdate();
    }
    con.commit();
catch (SqLException e){
    con.rollbackO;
    System.out.println(e.getMessage());
}
```

Figure 6.11 Inserting a Completed Order into the Database

DBDudes writes other JDBC code and stored procedures for all of the remaining tasks. They use code similar to some of the fragments that we have seen in this chapter.

- Establishing a connection to a database, as shown in Figure 6.2.
- Adding new books to the inventory, as shown in Figure 6.3.
- Processing results from SQL queries as shown in Figure 6.4-
- For each customer, showing how many orders he or she has placed. We showed a sample stored procedure for this query in Figure 6.8.
- Increasing the available number of copies of a book by adding inventory, as shown in Figure 6.9.
- Ranking customers according to their purchases, as shown in Figure 6.10.

DBDudcs takes care to make the application robust by processing exceptions and warnings, as shown in Figure 6.6.

DBDudes also decide to write a trigger, which is shown in Figure 6.12. Whenever a new order is entered into the Orders table, it is inserted with ship_date set to NULL. The trigger processes each row in the order and calls the stored procedure 'UpdateShipDate'. This stored procedure (whose code is not shown here) updates the (anticipated) ship_date of the new order to 'tomorrow', in case qtyjlLstock of the corresponding book in the Books table is greater than zero. Otherwise, the stored procedme sets the ship_date to two weeks.

```
CREATE TRIGGER update_ShipDate
        AFTER INSERT ON Orders                    /* Event */
FOR EACH ROW
BEGIN CALL UpdatcShipDate(new); END               /* Action */
```

Figure 6.12 Trigger to Update the Shipping Date of New Orders

6.7 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Why is it not straightforward to integrate SQL queries with a host programming language? (Section 6.1.1)
- How do we declare variables in Ernbcdded SQL? (Section 6.1.1)

- How do we use SQL statements within a host langl.lage? How do we check for errors in statement execution? (Section 6.1.1)
- Explain the impedance mismatch between host languages and SQL, and describe how cursors address this. (Section 6.1.2)
- What properties can cursors have? (Section 6.1.2)
- What is Dynamic SQL and how is it different from Embedded SQL? (Section 6.1.3)
- What is JDBC and what are its advantages? (Section 6.2)
- What are the components of the JDBC architecture? Describe four different architectural alternatives for JDBC drivers. (Section 6.2.1)
- How do we load JDBC drivers in Java code? (Section 6.3.1)
- How do we manage connections to data sources? What properties can connections have? (Section 6.3.2)
- What alternatives does JDBC provide for executing SQL DML and DDL statements? (Section 6.3.3)
- How do we handle exceptions and warnings in JDBC? (Section 6.3.5)
- 'What functionality provides the DatabaseMetaDataclass? (Section 6.3.6)
- What is SQLJ and how is it different from JDBC? (Section 6.4)
- Why are stored procedures important? How do we declare stored procedures and how are they called from application code? (Section 6.5)

EXERCISES

Exercise 6.1 Briefly answer the following questions.

1. Explain the following terms: Cursor, Embedded SQL, JDBC, SQLJ, stored procedure.
2. What are the differences between JDBC and SQLJ? \Nhy do they both exist?
3. Explain the term *stored procedure*, and give examples why stored procedures are useful.

Exercise 6.2 Explain how the following steps are performed in JDBC:

1. Connect to a data source.
2. Start, commit, and abort transactions.
3. Call a stored procedure.

How are these steps performed in SQLJ?

Exercise 6.3 Compare exception handling and handling of warnings ill embedded SQL, dynamic SQL, .IDBC, and SQL.I.

Exercise 6.4 Answer the following questions.

1. Why do we need a precompiler to translate embedded SQL and SQL.J? Why do we not need a precompiler for .IDBC?
2. SQL.J and embedded SQL use variables in the host language to pass parameters to SQL queries, whereas .JDBC uses placeholders marked with a '?'. Explain the difference, and why the different mechanisms are needed.

Exercise 6.5 A dynamic web site generates HTML pages from information stored in a database. Whenever a page is requested, is it dynamically assembled from static data and data in a database, resulting in a database access. Connecting to the database is usually a time-consuming process, since resources need to be allocated, and the user needs to be authenticated. Therefore, connection pooling--setting up a pool of persistent database connections and then reusing them for different requests can significantly improve the performance of database-backed websites. Since servlets can keep information beyond single requests, we can create a connection pool, and allocate resources from it to new requests.

Write a connection pool class that provides the following methods:

- Create the pool with a specified number of open connections to the database system.
- Obtain an open connection from the pool.
- Release a connection to the pool.
- Destroy the pool and close all connections.

PROJECT-BASED EXERCISES

In the following exercises, you will create database-backed applications. In this chapter, you will create the parts of the application that access the database. In the next chapter, you will extend this code to other aspects of the application. Detailed information about these exercises and material for more exercises can be found online at

http://www.cs.wisc.edu/-dbbook

Exercise 6.6 Recall the Notown Records database that you worked with in Exercise 2.5 and Exercise 3.15. You have now been tasked with designing a website for Notown. It should provide the following functionality:

- Users can search for records by name of the musician, title of the album, and Bame of the song.
- Users can register with the site, and registered users can log on to the site. Once logged on, users should not have to log on again unless they are inactive for a long time.
- Users who have logged on to the site can add items to a shopping basket.
- Users with items in their shopping basket can check out and make a purchase.

NOtOWIl wants to use JDBC to access the database. Write .JDBC code that performs the necessary data access and manipulation. You will integrate this code with application logic and presentation in the next chapter.

If Notown had chosen SQLJ instead of JDBC, how would your code change?

Exercise 6.7 Recall the database schema for Prescriptions-R-X that you created in Exercise 2.7. The Prescriptions-R-X chain of pharmacies has now engaged you to design their new website. The website has two different classes of users: doctors and patients. Doctors should be able to enter new prescriptions for their patients and modify existing prescriptions. Patients should be able to declare themselves as patients of a doctor; they should be able to check the status of their prescriptions online; and they should be able to purchase the prescriptions online so that the drugs can be shipped to their home address.

Follow the analogous steps from Exercise 6.6 to write JDBC code that performs the necessary data access and manipulation. You will integrate this code with application logic and presentation in the next chapter.

Exercise 6.8 Recall the university database schema that you worked with in Exercise 5.1. The university has decided to move enrollment to an online system. The website has two different classes of users: faculty and students. Faculty should be able to create new courses and delete existing courses, and students should be able to enroll in existing courses.

Follow the analogous steps from Exercise 6.6 to write JDBC code that performs the necessary data access and manipulation. You will integrate this code with application logic and presentation in the next chapter.

Exercise 6.9 Recall the airline reservation schema that you worked on in Exercise 5.3. Design an online airline reservation system. The reservation system will have two types of users: airline employees, and airline passengers. Airline employees can schedule new flights and cancel existing flights. Airline passengers can book existing flights from a given destination.

Follow the analogous steps from Exercise 6.6 to write JDBC code that performs the necessary data access and manipulation. You will integrate this code with application logic and presentation in the next chapter.

BIBLIOGRAPHIC NOTES

Information on ODBC can be found on Microsoft's web page (www.microsoft.com/data/odbc), and information on JDBC can be found on the Java web page (java. sun. com/products/jdbc). There exist rnany books on ODBC, for example, Sanders' ODBC Developer's Guicle [652] and the lvlicrosoft ODBC SDK [533]. Books on JDBC include works by Hamilton et al. [359], Reese [621], and White et a!. [773].

7

INTERNET APPLICATIONS

- How do we name resources on the Internet?
- How do Web browsers and webservers communicate?
- How do we present documents on the Internet? How do we differentiate between formatting and content?
- What is a three-tier application architecture? How do we write three-tiered applications?
- Why do we have application servers?
- Key concepts: Uniform Resource Identifiers (URI), Uniform Resource Locators (URL); Hypertext Transfer Protocol (HTTP), stateless protocol; Java; HTML; XML, XML DTD; three-tier architecture, client-server architecture; HTML forms; JavaScript; cascading style sheets, XSL; application server; Common Gateway Interface (CGI); servlet; JavaServer Page (JSP); cookie

Wow! They've got the Internet on computers now!

--Homer Simpson, *The Simpsons*

7.1 INTROpUCTION

The proliferation of computer networks, including the Internet and corporate 'intranets,' has enabled users to access a large number of data sources. This increased access to databases is likely to have a great practical impact; data and services can now be offered directly to customers in ways impossible until

recently. Examples of such electronic commerce applications include purchasing books through a \Veb retailer such as Amazon.com, engaging in online auctions at a site such as eBay, and exchanging bids and specifications for products between companies. The emergence of standards such as XrvIL for describing the content of documents is likely to further accelerate electronic commerce and other online applications.

While the first generation of Internet sites were collections of HTML files, most major sites today store a large part (if not all) of their data in database systems. They rely on DBMSs to provide fast, reliable responses to user requests received over the Internet. This is especially true of sites for electronic commerce and other business applications.

In this chapter, we present an overview of concepts that are central to Internet application development. We start out with a basic overview of how the Internet works in Section 7.2. We introduce HTML and XML, two data formats that are used to present data on the Internet, in Sections 7.3 and 7.4. In Section 7.5, we introduce three-tier architectures, a way of structuring Internet applications into different layers that encapsulate different functionality. In Sections 7.6 and 7.7, we describe the presentation layer and the middle layer in detail; the DBMS is the third layer. We conclude the chapter by discussing our B&N case study in Section 7.8.

Examples that appear in this chapter are available online at

http://www.cs.wisc.edu/-dbbook

7.2 INTERNET CONCEPTS

The Internet has emerged as a universal connector between globally distributed software systems. To understand how it works, we begin by discussing two basic issues: how sites on the Internet are identified, and how programs at one site communicate with other sites.

We first introduce Uniform Resource Identifiers, a naming schema for locating resources on the Internet in Section 7.2.1. \Ve then talk about the most popular protocol for accessing resources over the Web, the hypertext transfer protocol (HTTP) in Section 7.2.2.

7.2.1 Uniform Resource Identifiers

Uniform Resource Identifiers (URIs), are strings that uniquely identify resources 011 the Internet. A resource is any kind of information that can

Distributed Applications and Service-Oriented Architectures:

The advent of XML, due to its loosely-coupled nature, has made information exchange between different applications feasible to an extent previously unseen. By using XML for information exchange, applications can be written in different programming languages, run on different operating systems, and yet they can still share information with each other. There are also standards for externally describing the intended content of an XML file or message, most notably the recently adopted W3C XML Schemas standard.

A promising concept that has arisen out of the XML revolution is the notion of a Web service. A Web service is an application that provides a well-defined service, packaged as a set of remotely callable procedures accessible through the Internet. Web services have the potential to enable powerful new applications by *composing* existing Web services-all communicating seamlessly thanks to the use of standardizedXML-based information exchange. Several technologies have been developed or are currently under development that facilitate design and implementation of distributed applications. SOAP is a W3C standard for XML-based invocation of remote services (think XML RPC) that allows distributed applications to communicate either synchronously or asynchronously via structured, typed XML messages. SOAP calls can ride on a variety of underlying transport layers, including HTTP (part of what is making SOAP so successful) and various reliable messaging layers. Related to the SOAP standard are W3C's Web Services Description Language (WSDL) for describing Web service interfaces, and Universal Description, Discovery, and Integration (UDDI), a WSDL-based Web services registry standard (think yellow pages for Web services).

SOAP-based Web services are the foundation for Microsoft's recently released .NET framework, their application development infrastructure and associated run-time system for developing distributed applications, as well as for the Web services offerings of major software vendors such as IBM, BEA, and others. Many large software application vendors (major companies like PeopleSoft and SAP) have announced plans to provide Web service interfaces to their products and the data that they manage, and many are hoping that XML and Web services will finally provide the answer to the long-standing problem of enterprise application integration. Web services are also being looked to as a natural foundation for the next generation of business process management (or workflow) systems.

be identified by a URI, and examples include webpages, images, downloadable files, services that can be remotely invoked, mailboxes, and so on. The most common kind of resource is a static file (such as a HTML document), but a resource may also be a dynamically-generated HTML file, a movie, the output of a program, etc.

A URI has three parts:

- The (name of the) protocol used to access the resource.
- The host computer where the resource is located.
- The path name of the resource itself on the host computer.

Consider an example URI, such as http://www.bookstore.com/index.html. This URI can be interpreted as follows. Use the HTTP protocol (explained in the next section) to retrieve the document index.html located at the computer www.bookstore.com. This example URI is an instance of a Universal Resource Locator (URL), a subset of the more general URI naming scheme; the distinction is not important for our purposes. As another example, the following HTML fragment shows a URI that is an email address:

```
<a href=lImailto:webmaster@bookstore.com">Email the webmaster.</A>
```

7.2.2 The Hypertext Transfer Protocol (HTTP)

A communication protocol is a set of standards that defines the structure of messages between two communicating parties so that they can understand each other's messages. The Hypertext Transfer Protocol (HTTP) is the most common communication protocol used over the Internet. It is a client-server protocol in which a client (usually a Web browser) sends a request to an HTTP server, which sends a response back to the client. When a user requests a webpage (e.g., clicks on a hyperlink), the browser sends **HTTP** request messages for the objects in the page to the server. The server receives the requests and responds with **HTTP** response messages, which include the objects. It is important to recognize that HTTP is used to transmit all kinds of resources, not just files, but most resources on the Internet today are either static files or files output from server-side scripts.

A variant of the HTTP protocol called the Secure Sockets Layer (SSL) protocol uses encryption to exchange information securely between client and server. We postpone a discussion of SSL to Section 21.5.2 and present the basic HTTP protocol in this chapter.

As an example, consider what happens if a user clicks on the following link: http://www.bookstore.com/index.html. We first explain the structure of an HTTP request message and then the structure of an HTTP response message.

HTTP Requests

The client (Web browser) establishes a connection with the webserver that hosts the resource and sends a HTTP request message. The following example shows a sample HTTP request message:

```
GET index.html HTTP/1.1
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
```

The general structure of an HTTP request consists of several lines of ASCII text, with an empty line at the end. The first line, the request line, has three fields: the HTTP method field, the URI field, and the HTTP version field. The method field can take on values GET and POST; in the example the message requests the object index.html. (We discuss the differences between HTTP GET and HTTP POST in detail in Section 7.11.) The version field indicates which version of HTTP is used by the client and can be used for future extensions of the protocol. The user agent indicates the type of the client (e.g., versions of Netscape or Internet Explorer); we do not discuss this option further. The third line, starting with Accept, indicates what types of files the client is willing to accept. For example, if the page index.html contains a movie file with the extension .mpg, the server will not send this file to the client, as the client is not ready to accept it.

HTTP Responses

The server responds with an HTTP response message. It retrieves the page index.html, uses it to assemble the HTTP response message, and sends the message to the client. A sample HTTP response looks like this:

```
HTTP/1.1 200 OK
Date: Mon, 04 Mar 2002 12:00:00 GMT
Content-Length: 1024
Content-Type: text/html
Last-Modified: Mall, 22 JUll 1998 09:23:24 GMT
<HTML>
<HEAD>
</HEAD>
<BODY>
```

```
<H1>Barns and Nobble Internet Bookstore</H1>
Our inventory:
<H3>Science</H3>
<B>The Character of Physical Law</B>
```

The HTTP response message has three parts: a status line, several header lines, and the body of the message (which contains the actual object that the client requested). The status line has three fields (analogous to the request line of the HTTP request message): the HTTP version (HTTP/1.1), a status code (200), and an associated server message (OK). Common status codes and associated messages are:

- 200 OK: The request succeeded and the object is contained in the body of the response message";
- 400 Bad Request: A generic error code indicating that the request could not be fulfilled by the server.
- 404 Not Found: The requested object does not exist on the server.
- 505 HTTP Version Not Supported: The HTTP protocol version that the client uses is not supported by the server. (Recall that the HTTP protocol version sent in the client's request.)

Our example has three header lines: The date header line indicates the time and date when the HTTP response was created (not that this is not the object creation time). The Last-Modified header line indicates when the object was created. The Content-Length header line indicates the number of bytes in the object being sent after the last header line. The Content-Type header line indicates that the object in the entity body is HTML text.

The client (the Web browser) receives the response message, extracts the HTML file, parses it, and displays it. In doing so, it might find additional URIs in the file, and it then uses the HTTP protocol to retrieve each of these resources, establishing a new connection each time.

One important issue is that the HTTP protocol is a stateless protocol. Every message----from, the client to the HTTP server and vice-versa-is self-contained, and the connection established with a request is maintained only until the response message is sent. The protocol provides no mechanism to automatically 'remember' previous interactions between client and server.

The stateless nature of the HTTP protocol has a major impact on how Internet applications are written. Consider a user who interacts with our exalllple

bookstore application. Assume that the bookstore permits users to log into the site and then carry out several actions, such as ordering books or changing their address, without logging in again (until the login expires or the user logs out). How do we keep track of whether a user is logged in or not? Since HTTP is stateless, we cannot switch to a different state (say the 'logged in' state) at the protocol level. Instead, for every request that the user (more precisely, his or her Web browser) sends to the server, we must encode any *state* information required by the application, such as the user's login status. Alternatively, the server-side application code must maintain this state information and look it up on a per-request basis. This issue is explored further in Section 7.7.5.

Note that the statelessness of HTTP is a tradeoff between ease of implementation of the HTTP protocol and ease of application development. The designers of HTTP chose to keep the protocol itself simple, and deferred any functionality beyond the request of objects to application layers above the HTTP protocol.

7.3 HTML DOCUMENTS

In this section and the next, we focus on introducing HTML and XML. In Section 7.6, we consider how applications can use HTML and XML to create forms that capture user input, communicate with an HTTP server, and convert the results produced by the data management layer into one of these formats.

HTML is a simple language used to describe a document. It is also called a **markup language** because HTML works by augmenting regular text with 'marks' that hold special meaning for a Web browser. Commands in the language, called tags, consist (usually) of a start tag and an **end** tag of the form <TAG> and </TAG>, respectively. For example, consider the HTML fragment shown in Figure 7.1. It describes a webpage that shows a list of books. The document is enclosed by the tags <HTML> and </HTML>, marking it as an HTML document. The remainder of the document-enclosed in <BODY> ... </BoDY>-contains information about three books. Data about each book is represented as an unordered list (UL) whose entries are marked with the LI tag. HTML defines the set of valid tags as well as the meaning of the tags. For example, HTML specifies that the tag <TITLE> is a valid tag that denotes the title of the document. As another example, the tag <UL> always denotes an unordered list.

Audio, video, and even programs (written in Java, a highly portable language) can be included in HTML documents. When a user retrieves such a document using a suitable browser, images in the document arc displayed, audio and video clips are played, and embedded programs are executed at the user's machine; the result is a rich multimedia presentation. The case with which HTML docu-

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<Hl>Barns and Nobble Internet Bookstore</Hl>
Our inventory:
<H3>Science</H3>
    <B>The Character of Physical Law</B>
    <UL>
        <LI>Author: Richard Feynman</LI>
        <LI>Published 1980</LI>
        <LI>Hardcover</LI>
    </UL>
<H3>Fiction</H3>
    <B>Waiting for the Mahatma</B>
    <UL>
        <LI>Author: R.K. Narayan</LI>
        <LI>Published 1981</LI>
    </UL>
    <B>The English Teacher</B>
    <UL>
        <LI>Author: R.K. Narayan</LI>
        <LI>Published 1980</LI>
        <LI>Paperback</LI>
    </UL>
</BODY>
</HTML>
```

Figure 7.1 Book Listing in HTML

ments can be created—there are now visual editors that automatically generate HTML----and accessed using Internet browsers has fueled the explosive growth of the Web.

7.4 XML DOCUMENTS

In this section, we introduce XML as a document format, and consider how applications can utilize XML. Managing XML documents in a DBMS poses several new challenges; we discuss this aspect of XML in Chapter 27.

vVhile HTML can be used to mark up documents for display purposes, it is not adequate to describe the structure of the content for more general applications. For example, we can send the HTML document shown in Figure 7.1 to another application that displays it, but the second application cannot distinguish the first names of authors from their last names. (The application can try to recover such information by looking at the text inside the tags, but this defeats the purpose of using tags to describe document structure.) Therefore, HTML is unsuitable for the exchange of complex documents containing product specifications or bids, for example.

Extensible Markup Language (XML) is a markup language developed to remedy the shortcomings of HTML. In contrast to a fixed set of tags whose meaning is specified by the language (as in HTML), XML allows users to define new collections of tags that can be used to structure any type of data or document the user wishes to transmit. XML is an important bridge between the document-oriented view of data implicit in HTML and the schema-oriented view of data that is central to a DBMS. It has the potential to make database systems more tightly integrated into Web applications than ever before.

XML emerged from the confluence of two technologies, SGML and HTML. The Standard Generalized Markup Language (SGML) is a metalanguage that allows the definition of data and document interchange languages such as HTML. The SGML standard was published in 1988, and many organizations that rnanage a large number of complex documents have adopted it. Due to its generality, SGML is complex and requires sophisticated programs to harness its full potential. XML was developed to have much of the power of SGML while remaining relatively simple. Nonetheless, XML, like SGML, allows the definition of new document markup languages.

Although XML does not prevent a user from designing tags that encode the display of the data in a Web browser, there is a style language for XML called Extensible Style Language (XSL). XSL is a standard way of describing how an XML docmnent that adheres to a certain vocabulary of tags should be displayed.

7.4.1 Introduction to XML

We use the small XML docmnent shown in Figure 7.2 as an example.

- Elements: Elements, also called tags, arc the primary building blocks of an XML docmnent. The start of the content of an element ELM is marked with <ELM>, which is called the start tag, and the end of the content end is marked with </ELM>, called the end tag. In our example document.

The Design Goals of XML: XML was developed starting in 1996 by a working group under guidance of the World Wide Web Consortium (W3C) XML Special Interest Group. The design goals for XML included the following:

1. XML should be compatible with SGML.
2. It should be easy to write programs that process XML documents.
3. The design of XML should be formal and concise.

the element BOOKLIST encloses all information in the sample document. The element BOOK demarcates all data associated with a single book. XML elements are case sensitive: the element BOOK is different from Book. Elements must be properly nested. Start tags that appear inside the content of other tags must have a corresponding end tag. For example, consider the following XML fragment:

```
<BOOK>
    <AUTHOR>
        <FIRSTNAME>Richard</FIRSTNAME>
        <LASTNAME>Feynluan</LASTNAME>
    </AUTHOR>
</BOOK>
```

The element AUTHOR is completely nested inside the element BOOK, and both the elements LASTNAME and FIRSTNAME are nested inside the element AUTHOR.

- **Attributes:** An element can have descriptive attributes that provide additional information about the element. The values of attributes are set inside the start tag of an element. For example, let ELM denote an element with the attribute att. We can set the value of att to value through the following expression: <ELM att="value">. All attribute values must be enclosed in quotes. In Figure 7.2, the element BOOK has two attributes. The attribute GENRE indicates the genre of the book (science or fiction) and the attribute FORMAT indicates whether the book is a hardcover or a paperback.
- **Entity References:** Entities are shortcuts for portions of common text or the content of external files, and we call the usage of an entity in the XML document an entity reference. Wherever an entity reference appears in the document, it is textually replaced by its content. Entity references start with a '&' and end with a ';'. Five predefined entities in XML are placeholders for chara.cters with special meaning in XML. For example, the

```
<?xml version=11.0" encoding="UTF-S" standalone=llyesll?>
<BOOKLIST>
<BOOK GENRE=" Science" FORMAT=" Hardcover" >
    <AUTHOR>
        <FIRSTNAME>Richard</FIRSTNAME>
        <LASTNAME>Feynman</LASTNAME>
    </AUTHOR>
    <TITLE>The Character of Physical Law</TITLE>
    <PUBLISHED>1980</PUBLISHED>
</BOOK>
<BOOK GENRE=" Fiction" >
    <AUTHOR>
        <FIRSTNAME>R.K.</FIRSTNAME>
        <LASTNAME>Narayan</LASTNAME>
    </AUTHOR>
    <TITLE>Waiting for the Mahatma</TITLE>
    <PUBLISHED>1981</PUBLISHED>
</BOOK>
<BOOK GENRE=" Fiction" >
    <AUTHOR>
        <FIRSTNAME>R.K.</FIRSTNAME>
        <LASTNAME>Narayan</LASTNAME>
    </AUTHOR>
    <TITLE>The English Teacher</TITLE>
    <PUBLISHED> 1980</PUBLISHED>
</BOOK>
</BOOKLIST>
```

Figure 7.2 Book Information in XML

< character that marks the beginning of an XML command is reserved and has to be represented by the entity It. The other four reserved characters are &, >, ", and '; they are represented by the entities amp, gt, quot, and apos. For example, the text '1 < 5' has to be encoded in an XML document as follows: ' 1&lt ;5'. We can also use entities to insert arbitrary Unicode characters into the text. Unicode is a standard for character representations, similar to ASCII. For example, we can display the Japanese Hiragana character *a* using the entity reference &#x3042.

- Comments: We can insert comments anywhere in an XML document. Comments start with <!- and end with ->. Comments can contain arbitrary text except the string --.

- Document Type Declarations (DTDs): In XML, we can define our own markup language. A DTD is a set of rules that allows us to specify our own set of elements, attributes, and entities. Thus, a DTD is basically a grammar that indicates what tags are allowed, in what order they can appear, and how they can be nested. We discuss DTDs in detail in the next section.

We call an XML document well-formed if it has no associated DTD but follows these structural guidelines:

- The document starts with an XML declaration. An example of an XML declaration is the first line of the XML document shown in Figure 7.2.
- A root element contains all the other elements. In our example, the root element is the element BOOKLIST.
- All elements must be properly nested. This requirement states that start and end tags of an element must appear within the same enclosing element.

7.4.2 XML DTDs

A DTD is a set of rules that allows us to specify our own set of elements, attributes, and entities. A DTD specifies which elements we can use and constraints on these elements, for example, how elements can be nested and where elements can appear in the document. We call a document valid if a DTD is associated with it and the document is structured according to the rules set by the DTD. In the remainder of this section, we use the example DTD shown in Figure 7.3 to illustrate how to construct DTDs.

```
<!DOCTYPE BOOKLIST [
<! ELEMENT BOOKLIST (BOOK)*>
    <! ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?»
        <!ELEMENT AUTHOR (FIRSTNAME,LASTNAME»
            <! ELEMENT FIRSTNAME (#PCDATA»
            <! ELEMENT LASTNAME (#PCDATA»
        <! ELEMENT TITLE (#PCDATA»
        <!ELEMENT PUBLISHED (#PCDATA»
    <! ATTLIST BOOK GENRE (ScienceIFiction) #REQUIRED>
    <!ATTLIST BOOK FORMAT (PaperbackIHardcover) "Paperback">
]>
```

Figure 7.3 Bookstore XML DTD

A DTD is enclosed in <!DOCTYPE name [DTDdeclaration]>, where name is the name of the outermost enclosing tag, and DTDdeclaration is the text of the rules of the DTD. The DTD starts with the outermost element—the *root element*—which is BOOKLIST in our example. Consider the next rule:

```
<!ELEMENT BOOKLIST (BOOK)*>
```

This rule tells us that the element BOOKLIST consists of zero or more BOOK elements. The * after BOOK indicates how many BOOK elements can appear inside the BOOKLIST element. A * denotes zero or more occurrences, a + denotes one or more occurrences, and a? denotes zero or one occurrence. For example, if we want to ensure that a BOOKLIST has at least one book, we could change the rule as follows:

```
<!ELEMENT BOOKLIST (BOOK)+>
```

Let us look at the next rule:

```
<!ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?»
```

This rule states that a BOOK element contains a AUTHOR element, a TITLE element, and an optional PUBLISHED element. Note the use of the? to indicate that the information is optional by having zero or one occurrence of the element. Let us move ahead to the following rule:

```
< !ELEMENT LASTNAME (#PCDATA»
```

Until now we considered only elements that contained other elements. This rule states that LASTNAME is an element that does not contain other elements, but contains actual text. Elements that only contain other elements are said to have element content, whereas elements that also contain #PCDATA are said to have mixed content. In general, an element type declaration has the following structure:

```
<!ELEMENT (contentType»
```

Five possible content types are:

- Other elements.
- The special syrnbol #PCDATA, which indicates (parsed) character data.
- The special symbol EMPTY, which indicates that the element has no content. Elements that have no content are not required to have an end tag.
- The special symbol ANY, which indicates that any content is permitted. This content should be avoided whenever possible since it disables all checking of the document structure inside the element.

- A regular expression constructed from the preceding four choices. A regular expression is one of the following:
  - exp1, exp2, exp3: A list of regular expressions.
  - exp*: An optional expression (zero or more occurrences).
  - exp?: An optional expression (zero or one occurrences).
  - exp+: A mandatory expression (one or more occurrences).
  - expl | exp2: expl or exp2.

Attributes of elements are declared outside the element. For example, consider the following attribute declaration from Figure 7.3:

```
<! ATTLIST BOOK GENRE (ScienceIFiction) #REQUIRED»
```

This XML DTD fragment specifies the attribute GENRE, which is an attribute of the element BOOK. The attribute can take two values: Science or Fiction. Each BOOK element must be described in its start tag by a GENRE attribute since the attribute is required as indicated by #REQUIRED. Let us look at the general structure of a DTD attribute declaration:

```
<! ATTLIST elementName (attName attType default)+>
```

The keyword ATTLIST indicates the beginning of an attribute declaration. The string elementName is the name of the element with which the following attribute dcfinition is associated. What follows is the declaration of one or more attributes. Each attribute has a name, as indicated by attName, and a type, as indicated by attType. XML defines several possible types for an attribute. We discuss only string types and enumerated types here. An attribute of type string can take any string as a value. We can declare such an attribute by setting its type field to CDATA. For example, we can declare a third attribute of type string of the elernent BOOK as follows:

```
<!ATTLIST BOOK edition CDATA "1">
```

If an attribute has an enumerated type, we list all its possible values in the attribute declaration. In our example, the attribute GENRE is an enumerated attribute type; its possible attribute values are 'Science' and 'Fiction'.

The last part of an attribute declaration is called its default specification. The DTD in Figure 7.3 shows two different default specifications: #REQUIRED and the string 'Paperback'. The default specification #REQUIRED indicates that the attribute is required and whenever its associated element appears somewhere in the XML document a value for the attribute must be specified. The default specification indicated by the string 'Paperback' indicates that the attribute is not required; whenever its associated element appears without setting

```
<?xml version=11.0" encoding=IUTF-8" standalone="no"?>
<!DOCTYPE BOOKLIST SYSTEM" books.dtd" >
<BOOKLIST>
    <BOOK GENRE=" Science" FORMAT=" Hardcover" >
        <AUTHOR>
```

Figure 7.4 Book Information in XML

XML Schema: The DTD mechanism has several limitations, in spite of its widespread use. For example, elements and attributes cannot be assigned types in a flexible way, and elements are always ordered, even if the application does not require this. XML Schema is a new W3C proposal that provides a more powerful way to describe document structure than DTDs; it is a superset of DTDs, allowing legacy data to be handled easily. An interesting aspect is that it supports uniqueness and foreign key constraints.

a value for the attribute, the attribute automatically takes the value 'Paperback'. For example, we can make the attribute value 'Science' the default value for the GENRE attribute as follows:

```
<! ATTLIST BOOK GENRE (ScienceIFiction) "Science" >
```

In our bookstore example, the XML document with a reference to the DTD is shown in Figure 7.4.

7.4.3 Domain-Specific DTDs

Recently, DTDs have been developed for several specialized domains-including a wide range of commercial, engineering, financial, industrial, and scientific domains----and a lot of the excitement about XML has its origins in the belief that more and more standardized DTDs will be developed. Standardized DTDs would enable seamless data exchange among heterogeneous sources, a problem solved today either by implementing specialized protocols such as Electronic Data Interchange (EDI) or by implementing ad hoc solutions.

Even in an environment where all XML data is valid, it is not possible to straightforwardly integrate several XML documents by matching elements in their DTDs, because even when two elements have identical names in two different DTDs, the meaning of the elements could be completely different. If both documents use a single, standard DTD, we avoid this problem. The

development of standardized DTDs is more a social process than a research problem, since the major players in a given domain or industry segment have to collaborate.

For example, the mathematical markup language (MathML) has been developed for encoding mathematical material on the Web. There are two types of MathML elements. The 28 presentation elements describe the layout structure of a document; examples are the `mrow` element, which indicates a horizontal row of characters, and the msup element, which indicates a base and a subscript. The 75 content elements describe mathematical concepts. An example is the plus element, which denotes the addition operator. (A third type of element, the math element, is used to pass parameters to the MathML processor.) MathML allows us to encode mathematical objects in both notations since the requirements of the user of the objects might be different. Content elements encode the precise mathematical meaning of an object without ambiguity, and the description can be used by applications such as computer algebra systems. On the other hand, good notation can suggest the logical structure to a human and emphasize key aspects of an object; presentation elements allow us to describe mathematical objects at this level.

For example, consider the following simple equation:

$$x^2 - 4x - 32 = 0$$

Using presentation elements, the equation is represented as follows:

```
<mrow>
    <mrow> <msup><mi>x</mi><mn>2</mn></msup>
        <mo>-</mo>
        <mrow><mn>4</mn>
            <mo>&invisibletimes;</mo>
            <mi>x</mi>
        </mrow>
        <mo>-</mo><mn>32</mn>
    </mrow><mo>=</mo><mn>O</nm>
</mrow>
```

Using content elements, the equation is described as follows:

```
<reln><eq/>
    <apply>
        <minus/>
        <apply> <power/> <ci>x</ci> <cn>2</cn> </apply>
        <apply> <times/> <cn>4</cn> <ci>x</ci> </apply>
        <cn>32</cn>
```

```
        </apply> <cn>O</cn>
    </reln>
```

Note the additional power that we gain from using MathML instead of encoding the formula in HTML. The common way of displaying mathematical objects inside an HTML object is to include images that display the objects, for example, as in the following code fragment:

```
<IMG SRC=lIimages/equation.gifn ALI=II x**2 - 4x - 32 = 10 ll >
```

The equation is encoded inside an IMG tag with an alternative display format specified in the ALI tag. Using this encoding of a mathematical object leads to the following presentation problems. First, the image is usually sized to match a certain font size, and on systems with other font sizes the image is either too small or too large. Second, on systems with a different background color, the picture does not blend into the background and the resolution of the image is usually inferior when printing the document. Apart from problems with changing presentations, we cannot easily search for a formula or formula fragments on a page, since there is no specific markup tag.

7.5 THE THREE-TIER APPLICATION ARCHITECTURE

In this section, we discuss the overall architecture of data-intensive Internet applications. Data-intensive Internet applications can be understood in terms of three different functional components: *data management, application logic,* and *pTesentation.* The component that handles data mallgement usually utilizes a DBMS for data storage, but application logic and presentation involve much more than just the DBMS itself.

We start with a short overview of the history of database-backed application architectures, and introduce single-tier and client-server architectures in Section 7.5.1. We explain the three-tier architecture in detail in Section 7.5.2, and show its advantages in Section 7.5.3.

7.5.1 Single-Tier and Client-Server Architectures

In this section, we provide some perspective on the three-tier architecture by discussing single-tier and client-server architectures, the predecessors of the three-tier architecture. Initially, data-intensive applications were combined into a single tier, including the DBMS, application logic, and user interface, as illustrated in Figure 7.5. The application typically ran on a mainframe, and users accessed it through *dumb teT'minals* that could perform only data input and display. This approach has the benefit of being easily maintained by a central administrator.

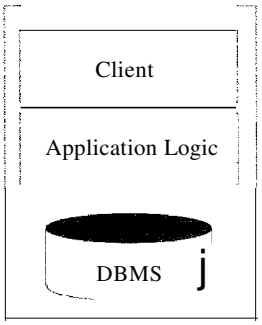

Figure 7.5 A Single-Tier Architecture

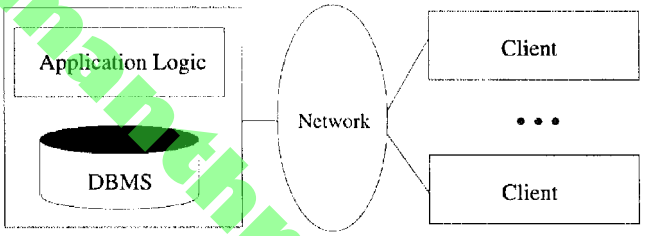

Figure 7.6 A Two-Server Architecture: Thin Clients

Single-tier architectures have an important drawback: Users expect graphical interfaces that require much more computational power than simple dumb terminals. Centralized computation of the graphical displays of such interfaces requires much more computational power than a single server has available, and thus single-tier architectures do not scale to thousands of users. The commoditization of the PC and the availability of cheap client computers led to the developlnent of the two-tier architecture.

Two-tier architectures, often also referred to as client-server architectures, consist of a client computer and a server computer, which interact through a well-defined protocol. What part of the functionality the client implements, and what part is left to the server, can vary. In the traditional client-server architecture, the client implements just the graphical user interface, and the server. implements both the business logic and the data management; such clients are often called thin clients, and this architecture is illustra,ted in Figure 7.6.

Other divisions are possible, such as more powerful clients that implement both user interface and business logic, or clients that implement user interface and part of the business logic, with the remaining part being implemented at the

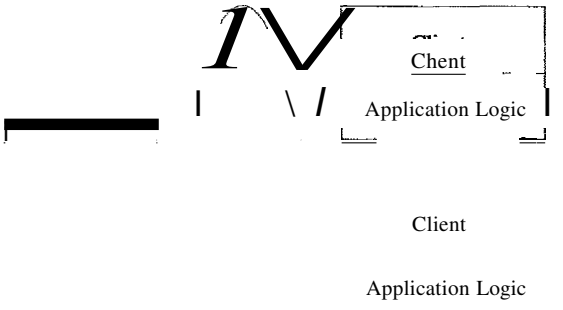

Figure 7.7 A Two-Tier Architecture: Thick Clients

server level; such clients are often called **thick** clients, and this architecture is illustrated in Figure 7.7.

Compared to the single-tier architecture, two-tier architectures physically separate the user interface from the data management layer. To implement two-tier architectures, we can no longer have dumb terminals on the client side; we require computers that run sophisticated presentation code (and possibly, application logic).

Over the last ten years, a large number of client-server development tools such Microsoft Visual Basic and Sybase Powerbuilder have been developed. These tools permit rapid development of client-server software, contributing to the success of the client-server model, especially the thin-client version.

The thick-client model has several disadvantages when compared to the thin-client model. First, there is no central place to update and maintain the business logic, since the application code runs at many client sites. Second, a large amount of trust is required between the server and the clients. As an exam-- pIe, the DBMS of a bank has to trust the (application executing at an) ATM machine to leave the database in a consistent state. (One way to address this problem is through *stored procedures*, trusted application code that is registered with the DBMS and can be called from SQL statelnents. We discuss stored procedures in detail in Section 6.5.)

A third disadvantage of the thick-client architecture is that it does not scale with the number of clients; it typically cannot handle more than a few hundred clients. The application logic at the client issues SQL queries to the server and the server returns the query result to the client, where further processing takes place. Large query results might be transferred between client and server.

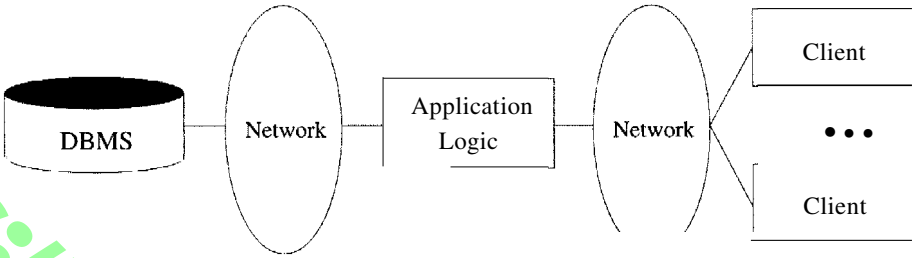

Figure 7.8 A Standard Three-Tier Architecture

(Stored procedures can mitigate this bottleneck.) Fourth, thick-client systems do not scale as the application accesses more and more database systems. Assume there are x different database systems that are accessed by y clients, then there are $x \cdot y$ different connections open at any time, clearly not a scalable solution.

These disadvantages of thick-client systems and the widespread adoption of standard, very thin clients—notably, Web browsers—have led to the widespread use thin-client architectures.

7.5.2 Three-Tier Architectures

The thin-client two-tier architecture essentially separates presentation issues from the rest of the application. The three-tier architecture goes one step further, and also separates application logic from data management:

- Presentation Tier: Users require a natural interface to make requests, provide input, and to see results. The widespread use of the Internet has made Web-based interfaces increasingly popular.
- Middle Tier: The application logic executes here. An enterprise-class application reflects complex business processes, and is coded in a general purpose language such as C++ or Java.
- Data Management Tier: Data-intensive Web applications involve DBMSs, which are the subject of this book.

Figure 7.8 shows a basic three-tier architecture. Different technologies have been developed to enable distribution of the three tiers of an application across multiple hardware platforms and different physical sites. Figure 7.9 shows the technologies relevant to each tier.

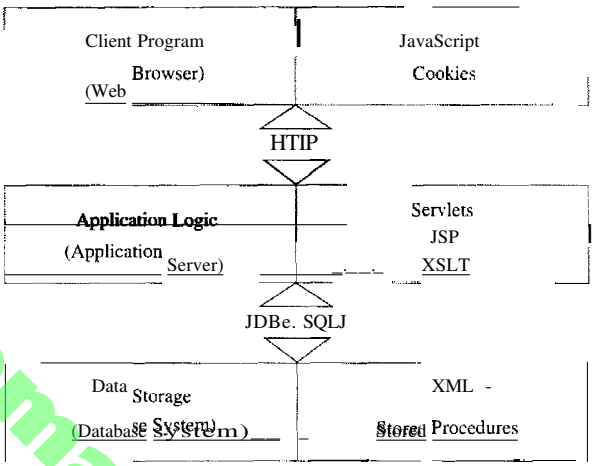

Figure 7.9 Technologies for the Three Tiers

Overview of the Presentation Tier

At the presentation layer, we need to provide forms through which the user can issue requests, and display responses that the middle tier generates. The hypertext markup language (HTML) discussed in Section 7.3 is the basic data presentation language.

It is important that this layer of code be easy to adapt to different display devices and formats; for example, regular desktops versus handheld devices versus cell phones. This adaptivity can be achieved either at the middle tier through generation of different pages for different types of client, or directly at the client through style sheets that specify how the data should be presented. In the latter case, the middle tier is responsible for producing the appropriate data in response to user requests, whereas the presentation layer decides *how* to display that information.

We cover presentation tier technologies, including style sheets, in Section 7.6.

Overview of the Middle Tier

The middle layer runs code that implements the business logic of the application: It controls what data needs to be input before an action can be executed, determines the control flow between multi-action steps, controls access to the database layer, and often assembles dynamically generated HTML pages from database query results.

The middle tier code is responsible for supporting all the different roles involved in the application. For example, in an Internet shopping site implementation, we would like customers to be able to browse the catalog and make purchases, administrators to be able to inspect current inventory, and possibly data analysts to ask summary queries about purchase histories. Each of these roles can require support for several complex actions.

For example, consider the a customer who wants to buy an item (after browsing or searching the site to find it). Before a sale can happen, the customer has to go through a series of steps: She has to add items to her shopping basket, she has to provide her shipping address and credit card number (unless she has an account at the site), and she has to finally confirm the sale with tax and shipping costs added. Controlling the flow among these steps and remembering already executed steps is done at the middle tier of the application. The data carried along during this series of steps might involve database accesses, but usually it is not yet permanent (for example, a shopping basket is not stored in the database until the sale is confirmed).

We cover the middle tier in detail in Section 7.7.

7.5.3 Advantages of the Three-Tier Architecture

The three-tier architecture has the following advantages:

- **Heterogeneous Systems:** Applications can utilize the strengths of different platforms and different software components at the different tiers. It is easy to modify or replace the code at any tier without affecting the other tiers.
- **Thin Clients:** Clients only need enough computation power for the presentation layer. Typically, clients are Web browsers.
- **Integrated Data Access:** In many applications, the data must be accessed from several sources. This can be handled transparently at the middle tier, where we can centrally manage connections to all database systems involved.
- **Scalability to Many Clients:** Each client is lightweight and all access to the system is through the middle tier. The middle tier can share database connections across clients, and if the middle tier becomes the bottle-neck, we can deploy several servers executing the middle tier code; clients can connect to anyone of these servers, if the logic is designed appropriately. This is illustrated in Figure 7.10, which also shows how the middle tier accesses multiple data sources. Of course, we rely upon the DBMS for each

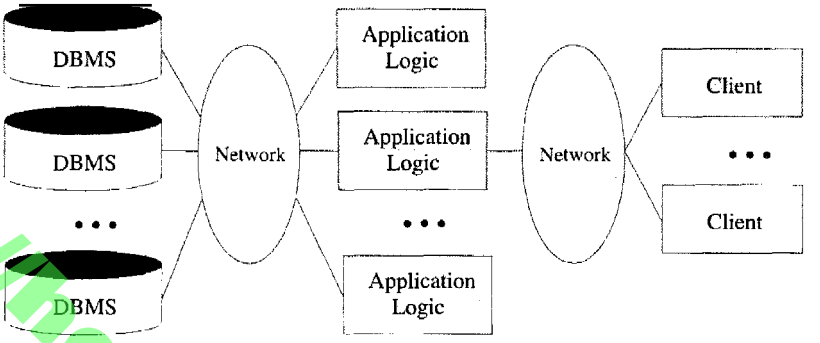

Figure 7.10 Middle-Tier Replication and Access to Multiple Data Sources

data source to be scalable (and this might involve additional parallelization or replication, as discussed in Chapter 22).

- Software Development Benefits: By dividing the application cleanly into parts that address presentation, data access, and business logic, we gain many advantages. The business logic is centralized, and is therefore easy to maintain, debug, and change. Interaction between tiers occurs through well-defined, standardized APIs. Therefore, each application tier can be built out of reusable components that can be individually developed, debugged, and tested.

7.6 THE PRESENTATION LAYER

In this section, we describe technologies for the client side of the three-tier architecture. We discuss HTML forms as a special means of passing arguments from the client to the middle tier (i.e., from the presentation tier to the middle tier) in Section 7.6.1. In Section 7.6.2, we introduce JavaScript, a Java-based scripting language that can be used for light-weight computation in the client tier (e.g., for simple animations). We conclude our discussion of client-side technologies by presenting style sheets in Section 7.6.3. Style sheets are languages that allow us to present the same webpage with different formatting for clients with different presentation capabilities; for example, Web browsers versus cell phones, or even a Netscape browser versus Microsoft's Internet Explorer.

7.6.1 HTML Forms

HTML forms are a common way of communicating data from the client tier to the middle tier. The general format of a form is the following:

```
<FORM ACTION="page.jsp" METHOD="GET" NAME="LoginForm">
```

```
</FORM>
```

A single HTML document can contain more than one form. Inside an HTML form, we can have any HTML tags except another FORM element.

The FORM tag has three important attributes:

- ACTION: Specifies the URI of the page to which the form contents are submitted; if the ACTION attribute is absent, then the URI of the current page is used. In the sample above, the form input would be submited to the page named page. j sp, which should provide logic for processing the input from the form. (We will explain methods for reading form data at the middle tier in Section 7.7.)
- METHOD: The HTTP/1.0 method used to submit the user input from the filled-out form to the webserver. There are two choices, GET and POST; we postpone their discussion to the next section.
- NAME: This attribute gives the form a name. Although not necessary, naming forms is good style. In Section 7.6.2, we discuss how to write client-side programs in JavaScript that refer to forms by name and perform checks on form fields.

Inside HTML forms, the INPUT, SELECT, and TEXTAREA tags are used to specify user input elements; a form can have many elements of each type. The simplest user input element is an INPUT field, a standalone tag with no terminating tag. An example of an INPUT tag is the following:

```
<INPUT TYPE=ltext" NAME="title">
```

The INPUT tag has several attributes. The three most important ones are TYPE, NAME, and VALUE. The TYPE attribute determines the type of the input field. If the TYPE attribute has value `text`, then the field is a text input field. If the TYPE attribute has value password, then the input field is a text field where the entered characters are displayed as stars on the screen. If the TYPE attribute has value `reset`, it is a simple button that resets all input fields within the form to their default values. If the TYPE attribute has value submit, then it is a button that sends the values of the different input fields in the form to the server. Note that `reset` and submit input fields affect the entire form.

The NAME attribute of the INPUT tag specifies the symbolic name for this field and is used to identify the value of this input field when it is sent to the server. NAME has to be set for INPUT tags of all types except submit and `reset`. In the preceding example, we specified `title` as the NAME of the input field.

The VALUE attribute of an input tag can be used for text or password fields to specify the default contents of the field. For submit or reset buttons, VALUE determines the label of the button.

The form in Figure 7.11 shows two text fields, one regular text input field and one password field. It also contains two buttons, a reset button labeled 'Reset Values' and a submit button labeled 'Log on.' Note that the two input fields are named, whereas the reset and submit button have no NAME attributes.

```
<FORM ACTION="page.jsp" METHoD="GET" NAME="LoginForm">
    <INPUT TYPE="text" NAME="username" VALUE="Joe"><P>
    <INPUT TYPE="password" NAME="p&ssword"><P>
    <INPUT TYPE="reset" VALUE="Reset Values"><P>
    <INPUT TYPE="submit" VALUE="Log on">
</FoRM>
```

Figure 7.11 HTML Form with Two Text Fields and Two Buttons

HTML forms have other ways of specifying user input, such as the aforementioned TEXTAREA and SELECT tags; we do not discuss them.

Passing Arguments to Server-Side Scripts

As mentioned at the beginning of Section 7.6.1, there are two different ways to submit HTML Form data to the webserver. If the method GET is used, then the contents of the form are assembled into a query URI (as discussed next) and sent to the server. If the method POST is used, then the contents of the form are encoded as in the GET method, but the contents are sent in a separate data block instead of appending them directly to the URI. Thus, in the GET method the form contents are directly visible to the user as the constructed URI, whereas in the POST method, the form contents are sent inside the HTTP request message body and are not visible to the user.

Using the GET method gives users the opportunity to bookmark the page with the constructed URI and thus directly jump to it in subsequent sessions; this is not possible with the POST method. The choice of GET versus POST should be determined by the application and its requirements.

Let us look at the encoding of the URI when the GET method is used. The encoded URI has the following form:

```
action?name1=value1&name2=value2&name3=value3
```

The action is the URI specified in the ACTION attribute to the FORM tag, or the current document URI if no ACTION attribute was specified. The 'name=value' pairs are the user inputs from the INPUT fields in the form. For form INPUT fields where the user did not input anything, the name is stil present with an empty value (name=). As a concrete example, consider the password submission form at the end of the previous section. Assume that the user inputs 'John Doe' as username, and 'secret' as password. Then the request URI is:

page.jsp?username=J01111+Doe&password=secret

The user input from forms can contain general ASCII characters, such as the space character, but URIs have to be single, consecutive strings with no spaces. Therefore, special characters such as spaces, '=', and other unprintable characters are encoded in a special way. To create a URI that has form fields encoded, we perform the following three steps:

1. Convert all special characters in the names and values to '%xyz,' where 'xyz' is the ASCII value of the character in hexadecimal. Special characters include =, &, %, +, and other unprintable characters. Note that we could encode *all* characters by their ASCII value.
2. Convert all space characters to the '+' character.
3. Glue corresponding names and values from an individual HTML INPUT tag together with '=' and then paste name-value pairs from different HTML INPUT tags together using'&' to create a request URI of the form: action?namel=value1&name2=value2&name3=value3

Note that in order to process the input elements from the HTML form at the middle tier, we need the ACTION attribute of the FORM tag to point to a page, script, or program that will process the values of the form fields the user entered. We discuss ways of receiving values from form fields in Sections 7.7.1 and 7.7.3.

7.6.2 JavaScript

JavaScript is a scripting language at the client tier with which we can add programs to webpages that run directly at the client (Le., at the machine running the Web browser). JavaScript is often used for the following types of computation at the client:

- Browser Detection: JavaScript can be used to detect the browser type and load a browser-specific page.
- Form Validation: JavaScript is used to perform simple consistency checks on form fields. For example, a JavaScript program might check whether a

form input that asks for an email address contains the character '@,' or if all required fields have been input by the user.

- **Browser Control:** This includes opening pages in customized windows; examples include the annoying pop-up advertisements that you see at many websites, which are programmed using JavaScript.

JavaScript is usually embedded into an HTML document with a special tag, the SCRIPT tag. The SCRIPT tag has the attribute LANGUAGE, which indicates the language in which the script is written. For JavaScript, we set the language attribute to JavaScript. Another attribute of the SCRIPT tag is the SRC attribute, which specifies an external file with JavaScript code that is automatically embedded into the HTML document. Usually JavaScript source code files use a '.js' extension. The following fragment shows a JavaScript file included in an HTML document:

```
<SCRIPT LANGUAGE="JavaScript" SRC="validateForm.js"> </SCRIPT>
```

The SCRIPT tag can be placed inside HTML comments so that the JavaScript code is not displayed verbatim in Web browsers that do not recognize the SCRIPT tag. Here is another JavaScipt code example that creates a pop-up box with a welcoming message. We enclose the JavaScipt code inside HTML comments for the reasons just mentioned.

```
<SCRIPT LANGUAGE="JavaScript">
<!--
    alert("Welcome to our bookstore");
//-->
</SCRIPT>
```

JavaScript provides two different commenting styles: single-line comments that start with the '//' character, and multi-line comments starting with '/*' and ending with '*/' characters.[1]

JavaScript has variables that can be numbers, boolean values (true or false), strings, and some other data types that we do not discuss. Global variables have to be declared in advance of their usage with the keyword var, and they can be used anywhere inside the HTML documents. Variables local to a JavaScript function (explained next) need not be declared. Variables do not have a fixed type, but implicitly have the type of the data to which they have been assigned.

[1] Actually, '<!--' also marks the start of a single-line comment, which is why we did not have to mark the HTML starting cormnent '<!--' in the preceding example using JavaScript comment notation. In contrast, the HTML closing comment "-->" has to be commented out in JavaScript as it is interpreted otherwise.

JavaScript has the usual assignment operators (=, + =, etc.), the usual arithmetic operators (+, -, *, /, %), the usual comparison operators (==, ! =, >=, etc.), and the usual boolean operators (&& for logical AND, || for logical OR, and! for negation). Strings can be concatenated using the '+' character. The type of an object determines the behavior of operators; for example 1+1 is 2, since we are adding numbers, whereas "1"+"1" is "11," since we are concatenating strings. JavaScript contains the usual types of statements, such as assignments, conditional statements (if Condition) {statements;} else {statements; }), and loops (for-loop, do-while, and while-loop).

JavaScript allows us to create functions using the function keyword: function f Cargl, arg2) {statements;}. We can call functions from JavaScript code, and functions can return values using the keyword return.

We conclude this introduction to JavaScript with a larger example of a JavaScript function that tests whether the login and password fields of a HTML form are not empty. Figure 7.12 shows the JavaScript function and the HTML form. The JavaScript code is a function called testLoginEmptyO that tests whether either of the two input fields in the form named LoginForm is empty. In the function testLoginEmpty, we first use variable loginForm to refer to the form LoginForm using the implicitly defined variable document, which refers to the current HTML page. (JavaScript has a library of objects that are implicitly defined.) We then check whether either of the strings loginForm. userif. value or loginForm. password. value is empty.

The function testLoginEmpty is checked within a form event handler. An event **handler** is a function that is called if an event happens on an object in a webpage. The event handler we use is onSubmit, which is called if the submit button is pressed (or if the user presses return in a text field in the form). If the event handler returns true, then the form contents are submitted to the server, otherwise the form contents are not submitted to the server.

JavaScript has functionality that goes beyond the basics that we explained in this section; the interested reader is referred to the bibliographic notes at the end of this chapter.

7.6.3 Style Sheets

Different clients have different displays, and we need correspondingly different ways of displaying the same information. For example, in the simplest case, we might need to use different font sizes or colors that provide high-contrast on a black-and-white screen. As a more sophisticated example, we might need to re-arrange objects on the page to accommodate small screens in personal

```
<SCRIPT LANGUAGE="JavaScript">
<!--
function testLoginEmpty()
{
    10ginForm = document.LoginForm
    if ((loginForm.userid.value == "") ||
            (loginFonn.password.value == |.|)) {
        alert(,Please enter values for userid and password.');
        return false;
    }
    else
        return true;
}
//-->
</SCRIPT>
<Hi ALIGN = "CENTER">Barns and Nobble Internet Bookstore</Hi>
<H3 ALIGN = "CENTER">Please enter your userid and password:</H3>
<FORM NAME = "LoginForm" METHOD="POST"
        ACTION="TableOfContents.jsp"
        onSubmit="return testLoginEmptyO">
    Userid: <INPUT TYPE="TEXT" NAME="userid"><P>
    Password: <INPUT TYPE="PASSWORD" NAME="password"><P>
    <INPUT TYPE="SUBMIT" VALUE="Login" NAME="SUBMIT">
    <INPUT TYPE="RESET" VALUE="Clear Input" NAME="RESET">
</FORM>
```

Figure 7.12 Form Validation with JavaScript

digital assistants (PDAs). As another example, we might highlight different infonnation to focus on some important part of the page. A style sheet is a method to adapt the same document contents to different presentation formats. A style sheet contains instructions that tell a Web browser (or whatever the client uses to display the webpage) how to translate the data of a document into a presentation that is suitable for the client's display.

Style sheets separate the transformative aspect of the page from the rendering aspects of the page. During transformation, the objects in the XML document are rearranged to form a different structure, to omit parts of the XML document, or to merge two different XML documents into a single document. During rendering, we take the existing hierarchical structure of the XML document and format the document according to the user's display device.

```
BODY {BACKGROUND-COLOR: yellow}
Hi {FONT-SIZE: 36pt}
H3 {COLOR: blue}
P {MARGIN-LEFT: 50px; COLOR: red}
```

Figure 7.13 An Example Style sheet

The use of style sheets has many advantages. First, we can reuse the same document many times and display it differently depending on the context. Second, we can tailor the display to the reader's preference such as font size, color style, and even level of detail. Third, we can deal with different output formats, such as different output devices (laptops versus cell phones), different display sizes (letter versus legal paper), and different display media (paper versus digital display). Fourth, we can standardize the display format within a corporation and thus apply style sheet conventions to documents at any time. Further, changes and improvements to these display conventions can be managed at a central place.

There are two style sheet languages: XSL and CSS. CSS was created for HTML with the goal of separating the display characteristics of different formatting tags from the tags themselves. XSL is an extension of CSS to arbitrary XML docurnents; besides allowing us to define ways of formatting objects, XSL contains a transformation language that enables us to rearrange objects. The target files for CSS are HTML files, whereas the target files for XSL are XML files.

Cascading Style Sheets

A Cascading Style Sheet (CSS) defines how to display HTML elements. (In Section 7.13, we introduce a more general style sheet language designed for XML documents.) Styles are normally stored in style sheets, which are files that contain style definitions. Many different HTML documents, such as all documents in a website, can refer to the same CSS. Thus, we can change the format of a website by changing a single file. This is a very convenient way of changing the layout of many webpages at the same time, and a first step toward the separation of content from presentation.

An example style sheet is shown in Figure 7.13. It is included into an HTML file with the following line:

```
<LINK REL="style sheet" TYPE="text/css" HREF="books.css" />
```

Each line in a CSS sheet consists of three parts; a selector, a property, and a value. They are syntactically arranged in the following way:

```
selector {property: value}
```

The selector is the element or tag whose format we are defining. The property indicates the tag's attribute whose value we want to set in the style sheet, and the property is the actual value of the attribute. As an example, consider the first line of the example style sheet shown in Figure 7.13:

```
BODY {BACKGROUND-COLOR: yellow}
```

This line has the same effect as changing the HTML code to the following:

```
<BODY BACKGROUND-COLOR="yellow">.
```

The value should always be quoted, as it could consist of several words. More than one property for the same selector can be separated by semicolons as shown in the last line of the example in Figure 7.13:

```
P {MARGIN-LEFT: 50px; COLOR: red}
```

Cascading style sheets have an extensive syntax; the bibliographic notes at the end of the chapter point to books and online resources on CSSs.

XSL

XSL is a language for expressing style sheets. An XSL style sheet is, like CSS, a file that describes how to display an XML document of a given type. XSL shares the functionality of CSS and is compatible with it (although it uses a different syntax).

The capabilities of XSL vastly exceed the functionality of CSS. XSL contains the XSL Transformation language, or XSLT, a language that allows us to transform the input XML document into a XML document with another structure. For example, with XSLT we can change the order of elements that we are displaying (e.g.; by sorting them), process elements more than once, suppress elements in one place and present them in another, and add generated text to the presentation.

XSL also contains the XML Path Language (XPath), a language that allows us to refer to parts of an XML document. We discuss XPath in Section

27. XSL also contains XSL Formatting Object, a way of formatting the output of an XSL transformation.

7.7 THE MIDDLE TIER

In this section, we discuss technologies for the middle tier. The first generation of middle-tier applications were stand-alone programs written in a general-purpose programming language such as C, C++, and Perl. Programmers quickly realized that interaction with a stand-alone application was quite costly; the overheads include starting the application every time it is invoked and switching processes between the webserver and the application. Therefore, such interactions do not scale to large numbers of concurrent users. This led to the development of the application server, which provides the run-time environment for several technologies that can be used to program middle-tier application components. Most of today's large-scale websites use an application server to run application code at the middle tier.

Our coverage of technologies for the middle tier mirrors this evolution. We start in Section 7.7.1 with the Common Gateway Interface, a protocol that is used to transmit arguments from HTML forms to application programs running at the middle tier. We introduce application servers in Section 7.7.2. We then describe technologies for writing application logic at the middle tier: Java servlets (Section 7.7.3) and Java Server Pages (Section 7.7.4). Another important functionality is the maintenance of state in the middle tier component of the application as the client component goes through a series of steps to complete a transaction (for example, the purchase of a market basket of items or the reservation of a flight). In Section 7.7.5, we discuss Cookies, one approach to maintaining state.

7.7.1 CGI: The Common Gateway Interface

The Common Gateway Interface connects HTML forms with application programs. It is a protocol that defines how arguments from forms are passed to programs at the server side. We do not go into the details of the actual CGI protocol since libraries enable application programs to get arguments from the HTML fonn; we shortly see an example in a CGI program. Programs that communicate with the webserver via CGI are often called CGI scripts, since many such application programs were written in a scripting language such as Perl.

As an example of a program that interfaces with an HTML form via CGI, consider the sample page shown in Figure 7.14. This webpage contains a form where a user can fill in the name of an author. If the user presses the 'Send

```
<HTML><HEAD><TITLE>The Database Bookstore</TITLE></HEAD>
<BODY>
<FORM ACTION="find_books.cgi" METHOD=POST>
    Type an author name:
    <INPUT TYPE="text" NAME=lauthorName"
           SIZE=30 MAXLENGTH=50>
    <INPUT TYPE="submitil value="Send it">
    <INPUT TYPE=lreset" VALUE="Clear form">
</FORM>
</BODY></HTML>
```

Figure 7.14 A Sample Web Page Where Form Input Is Sent to a CGI Script

it' button, the Perl script 'findBooks.cgi' shown in Figure 7.14 is executed as a separate process. The CGl protocol defines how the communication between the form and the script is performed. Figure 7.15 illustrates the processes created when using the CGl protocol.

Figure 7.16 shows the example CGl script, written in Perl. We omit error-checking code for simplicity. Perl is an interpreted language that is often used for CGl scripting and many Perl libraries, called **modules,** provide high-level interfaces to the CGl protocol. \Ve use one such library, called the **DBI library,** in our example. The CGI module is a convenient collection of functions for creating CGl scripts. In part 1 of the sample script, we extract the argument of the HTML form that is passed along from the client as follows:

```
$authorName = $dataIn->paramCauthorName');
```

Note that the parameter name authorName was used in the form in Figure 7.14 to name the first input field. Conveniently, the CGl protocol abstracts the actual implementation of how the webpage is returned to the Web browser; the webpage consists simply of the output of our program, and we start assembling the output HTML page in part 2. Everything the script writes in print-statements is part of the dynamically constructed webpage returned to the browser. We finish in part 3 by appending the closing format tags to the resulting page.

7.7.2 Application Servers

Application logic can be enforced through server-side programs that are invoked using the CGl protocol. However, since each page request results in the creation of a new process, this solution does not scale well to a large number of simultaneous requests. This performance problem led to the development of

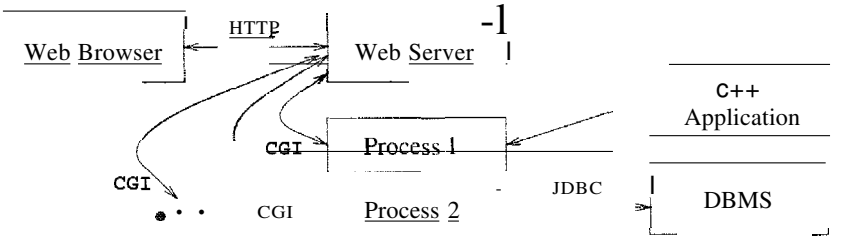

Figure 7.15 Process Structure with eGI Scripts

```
#!/usr/bin/perl
use CGI;

### part 1
$dataIn = new CGI;
$dataIn-¿header();
$authorName = $dataIn-l,param('authorName');

### part 2
print (II<HTML><TITLE>Argument passing test</TITLE> II) ;
print (II The user passed the following argument: II) ;
print (II authorName: ", $authorName);

### part 3
print ("</HTML>");
exit;
```

Figure 7.16 A Simple Perl Script

specialized programs called application servers. An application server maintains a pool of threads or processes and uses these to execute requests. Thus, it avoids the startup cost of creating a new process for each request.

Application servers have evolved into flexible middle-tier packages that provide many functions in addition to eliminating the process-creation overhead. They facilitate concurrent access to several heterogeneous data sources (e.g., by providing JDBC drivers), and provide session management services. Often, business processes involve several steps. Users expect the system to maintain continuity during such a multistep session. Several session identifiers such as cookies, URI extensions, and hidden fields in HTML forms can be used to identify a session. Application servers provide functionality to detect when a session starts and ends and keep track of the sessions of individual users. They

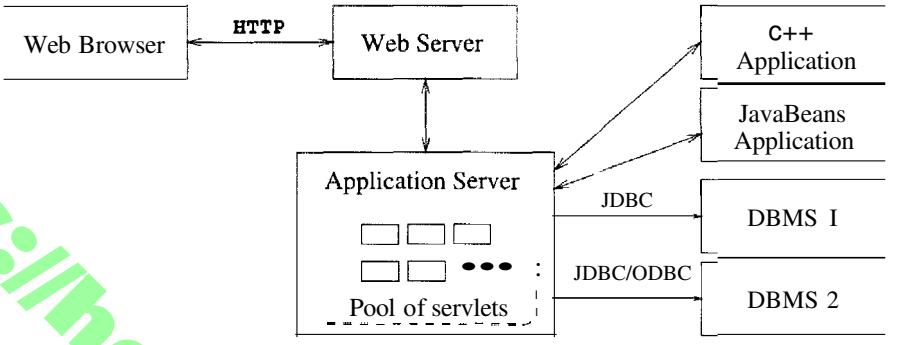

Figure 7.17 Process Structure in the Application Server Architecture

also help to ensure secure database access by supporting a general user-id mechanism. (For more on security, see Chapter 21.)

A possible architecture for a website with an application server is shown in Figure 7.17. The client (a Web browser) interacts with the webserver through the HTTP protocol. The webserver delivers static HTML or XML pages directly to the client. To assemble dynamic pages, the webserver sends a request to the application server. The application server contacts one or more data sources to retrieve necessary data or sends update requests to the data sources. After the interaction with the data sources is completed, the application server assembles the webpage and reports the result to the webserver, which retrieves the page and delivers it to the client.

The execution of business logic at the webserver's site, server-side processing, has become a standard model for implementing more complicated business processes on the Internet. There are many different technologies for server-side processing and we only mention a few in this section; the interested reader is referred to the bibliographic notes at the end of the chapter.

7.7.3 Servlets

Java servlets are pieces of Java code that run on the middle tier, in either webservers or application servers. There are special conventions on how to read the input from the user request and how to write output generated by the servlet. Servlets are truly platform-independent, and so they have become very popular with Web developers.

Since servlets are Java programs, they are very versatile. For example, servlets can build webpages, access databases, and maintain state. Servlets have access

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

pUblic class ServletTemplate extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        // Use 'out' to send content to browser
        out.println("Hello World");
    }
}
```

Figure 7.18 Servlet Template

to all Java APIs, including JDBC. All servlets must implement the Servlet interface. In most cases, servlets extend the specific HttpServlet class for servers that communicate with clients via HTTP. The HttpServlet class provides methods such as doGet and doPost to receive arguments from HTML forms, and it sends its output back to the elient via HTTP. Servlets that communicate through other protocols (such as ftp) need to extend the class GenericServlet.

Servlets are compiled Java classes executed and maintained by a servlet container. The servlet container manages the lifespan of individual servlets by creating and destroying them. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by webservers. For such applications, there is a useful library of HTTP-specific servlet classes.

Servlets usually handle requests from HTML forms and maintain state between the client and the server. We discuss how to maintain state in Section 7.7.5. A template of a generic servlet structure is shown in Figure 7.18. This simple servlet just outputs the two words "Hello World," but it shows the general structure of a full-fledged servlet. The request object is used to read HTML form data. The response object is used to specify the HTTP response status code and headers of the HTTP response. The object out is used to compose the content that is returned to the client.

Recall that HTTP sends back the status line, a header, a blank line, and then the context. Right now our servlet just returns plain text. We can extend our servlet by setting the content type to HTML, generating HTML as follows:

```
PrinfWriter out = response.get\Vriter();
String docType =
    "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
    "Transitional//EN"> \n";
out.println(docType +
    "<HTML>\n" +
    "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
    "<BODY>\n" +
    "<Hl>Hello WWW</Hl>\n" +
    "</BODY></HTML>");
```

What happens during the life of a servlet? Several methods are called at different stages in the development of a servlet. When a requested page is a servlet, the webserver forwards the request to the servlet container, which creates an instance of the servlet if necessary. At servlet creation time, the servlet container calls the init () method, and before deallocating the servlet, the servlet container calls the servlet's destroyO method.

When a servlet container calls a servlet because of a requested page, it starts with the service () method, whose default behavior is to call one of the following methods based on the HTTP transfer method: service () calls doGet 0 for a HTTP GET request, and it calls doPost () for a HTTP POST request. This automatic dispatching allows the servlet to perform different tasks on the request data depending on the HTTP transfer method. Usually, we do not override the service () method, unless we want to program a servlet that handles both HTTP POST and HTTP GET requests identically.

We conclude our discussion of servlets with an example, shown in Figure 7.19, that illustrates how to pass arguments from an HTML form to a servlet.

7.7.4 JavaServer Pages

In the previous section, we saw how to use Java programs in the middle tier to encode application logic and dynamically generate webpages. If we needed to generate HTML output, we wrote it to the out object. Thus, we can think about servlets as Java code embodying application logic, with embedded HTML for output.

JavaServer pages (JSPs) interchange the roles of output and application logic. JavaServer pages are written in HTML with servlet-like code embedded in special HTlVIL tags. Thus, in comparison to servlets, JavaServer pages are better suited to quickly building interfaces that have some logic inside, whereas servlets are better suited for complex application logic.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ReadUserName extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType('j textjhtml'j);
        PrintWriter out = response.getWriter();

        out.println("<BODY>\n" +
            "<Hi ALIGN=CENTER> Username: </Hi>\n" +
            "<UL>\n" +
            " <LI>title: "
            + request.getParameter("userid") + "\n" +
            + request.getParameter("password'j) + "\n" +
            1</UL>\n" +
            1</BODY></HTML>")j
    }

    public void doPost(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Figure 7.19 Extracting the User Name and Password From a Form

While there is a big difference for the programmer, the middle tier handles JavaServer pages in a very simple way: They are usually compiled into a servlet, which is then handled by a servlet container analogous to other servlets.

The code fragment in Figure 7.20 shows a simple JSP example. In the middle of the HTML code, we access information that was passed from a form.

```
<!DOCTYPE HTML PUBLIC 11_//W3C//DTD HTML 4.0
        Transitional//EN‖>
<HTML>
<HEAD><TITLE>Welcome to Barnes and Nobble</TITLE></HEAD>
<BODY>
    <Hl>Welcome back!</Hl>
    <% String name="NewUser‖;
        if (request.getParameter(lIusernamell) != null) {
            name=request.getParameter("username");
        }
    %>
    You are logged on as user <%=name%>
    <P>
    Regular HTML for all the rest of the on-line store's webpage.
</BODY>
</HTML>
```

Figure 7.20 Reading Form Parameters in JSP

7.7.5 Maintaining State

As discussed in previous sections, there is a need to maintain a user's state across different pages. As an example, consider a user who wants to make a purchase at the Barnes and Nobble website. The user must first add items into her shopping basket, which persists while she navigates through the site. Thus, we use the notion of state mainly to remember information as the user navigates through the site.

The HTTP protocol is stateless. We call an interaction with a webserver stateless if no inforination is retained from one request to the next request. We call an interaction with a webserver stateful, or we say that state is maintained, if some memory is stored between requests to the server, and different actions are taken depending on the contents stored.

In our example of Barnes and Nobble, we need to maintain the shopping basket of a user. Since state is not encapsulated in the HTTP protocol, it has to be maintained either at the server or at the client. Since the HTTP protocol is stateless by design, let us review the advantages and disadvantages of this design decision. First, a stateless protocol is easy to program and use, and it is great for applications that require just retrieval of static information. In addition, no extra memory is used to maintain state, and thus the protocol itself is very efficient. On the other hand, without some additional mechanism at the presentation tier and the middle tier, we have no record of previous requests, and we cannot program shopping baskets or user logins.

Since we cannot maintain state in the HTTP protocol, where should we mtain-tain state? There are basically two choices. We can maintain state in the middle tier, by storing information in the local main memory of the applica-tion logic, or even in a database system. Alternatively, we can maintain state on the client side by storing data in the form of a *cookie.* We discuss these two ways of maintaining state in the next two sections.

Maintaining State at the Middle Tier

At the middle tier, we have several choices as to *where* we maintain state. First, we could store the state at the bottom tier, in the database server. The state survives crashes of the system, but a database access is required to query or update the state, a potential performance bottleneck. An alternative is to store state in main memory at the middle tier. The drawbacks are that this information is volatile and that it might take up a lot of main memory. We can also store state in local files at the middle tier, as a compromise between the first two approaches.

A rule of thumb is to use state maintenance at the middle tier or database tier only for data that needs to persist over many different user sessions. Examples of such data are past customer orders, click-stream data recording a user's movement through the website, or other permanent choices that a user makes, such as decisions about personalized site layout, types of messages the user is willing to receive, and so on. As these examples illustrate, state information is often centered around users who interact with the website.

Maintaining State at the Presentation Tier: Cookies

Another possibility is to store state at the presentation tier and pass it to the middle tier with every HTTP request. We essentially work around around the statelessness of the HTTP protocol by sending additional information with every request. Such information is called a cookie.

A **cookie** is a collection of *(name, value)*-pairs that can be manipulated at the presentation and middle tiers. Cookies are easy to use in Java servlets and Java8erver Pages and provide a simple way to make non-essential data persistent at the client. They survive several client sessions because they persist in the browser cache even after the browser is closed.

One disadvantage of cookies is that they are often perceived as as being invasive, and many users disable cookies in their Web browser; browsers allow users to prevent cookies from being saved on their machines. Another disadvantage is that the data in a cookie is currently limited to 4KB, but for most applications this is not a bad limit.

We can use cookies to store information such as the user's shopping basket, login information, and other non-permanent choices made in the current session.

Next, we discuss how cookies can be manipulated from servlets at the middle tier.

The Servlet Cookie API

A cookie is stored. in a small text file at the client and. contains *(name, value)*-pairs, where both name and value are strings. We create a new cookie through the Java Cookie class in the middle tier application code:

```
Cookie cookie = new Cookie("username","guest");
cookie.setDomain("www.bookstore.com");
cookie.set8ecure(false);                    // no 88L required
cookie.setMaxAge(60*60*24*7*31);      // one month lifetime
response.addCookie(cookie);
```

Let us look at each part of this code. First, we create a new Cookie object with the specified *(name, value)*-pair. Then we set attributes of the cookie; we list some of the most common attributes below:

- setDomain and getDomain: The domain specifies the website that will receive the cookie. The default value for this attribute is the domain that created the cookie.
- setSecure and getSecure: If this flag is true, then the cookie is sent only if we are llsing a secure version of the HTTP protocol, such as 88L.
- setMaxAge and getMaxAge: The MaxAge attribute determines the lifetime of the cookie in seconds. If the value of MaxAge is less than or equal to zero, the cookie is deleted when the browser is closed.

- setName and getName: We did not use these functions in our code fragment; they allow us to Ilame the cookie.
- setValue and getValue: These functions allow us to set and read the value of the cookie.

The cookie is added to the request object within the Java servlet to be sent to the client. Once a cookie is received from a site (www.bookstore.comin this example), the client's Web browser appends it to all HTTP requests it sends to this site, until the cookie expires.

We can access the contents of a cookie in the middle-tier code through the request object getCookies 0 method, which returns an array of Cookie objects. The following code fragment reads the array and looks for the cookie with name 'username.'

```
Cookie[] cookies = request.getCookiesO;
String theUser;
for(int i=O; i < cookies.length; i++) {
     Cookie cookie = cookies[i];
     if (cookie.getNameO.equals("username"))
          theUser = cookie.getValueO;
}
```

A simple test can be used to check whether the user has turned off cookies: Send a cookie to the user, and then check whether the request object that is returned still contains the cookie. Note that a cookie should never contain an unencrypted password or other private, unencrypted data, as the user can easily inspect, modify, and erase any cookie at any time, including in the middle of a session. The application logic needs to have sufficient consistency checks to ensure that the data in the cookie is valid.

7.8 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes now moves on to the implementation of the application layer and considers alternatives for connecting the DBMS to the World Wide Web.

DBDudes begifls by considering session management. For example, users who log in to the site, browse the catalog, and select books to buy do not want to re-enter their cllstomer identification numbers. Session management has to extend to the whole process of selecting books, adding them to a shopping cart, possibly removing books from the cart, and checking out and paying for the books.

DBDudes then considers whether webpages for books should be static or dynamic. If there is a static webpage for each book, then we need an extra database field in the Books relation that points to the location of the file. Even though this enables special page designs for different books, it is a very labor-intensive solution. DBDudes convinces B&N to dynamically assemble the webpage for a book from a standard template instantiated with information about the book in the Books relation. Thus, DBDudes do not use static HTML pages, such as the one shown in Figure 7.1, to display the inventory.

DBDudes considers the use of XML as a data exchange format between the database server and the middle tier, or the middle tier and the client tier. Representation of the data in XML at the middle tier as shown in Figures 7.2 and 7.3 would allow easier integration of other data sources in the future, but B&N decides that they do not anticipate a need for such integration, and so DBDudes decide not to use XML data exchange at this time.

DBDudes designs the application logic as follows. They think that there will be four different webpages:

- index.jsp: The home page of Barns and Nobble. This is the main entry point for the shop. This page has search text fields and buttons that allow the user to search by author name, ISBN, or title of the book. There is also a link to the page that shows the shopping cart, cart.jsp.
- login.jsp: Allows registered users to log in. Here DBDudes use an HTML form similar to the one displayed in Figure 7.11. At the middle tier, they use a code fragment similar to the piece shown in Figure 7.19 and JavaServerPages as shown in Figure 7.20.
- search.jsp: Lists all books in the database that match the search condition specified by the user. The user can add listed items to the shopping basket; each book has a button next to it that adds it. (If the item is already in the shopping basket, it increments the quantity by one.) There is also a counter that shows the total number of items currently in the shopping basket. (DBDucles makes a note that that a quantity of five for a single item in the shopping basket should indicate a total purchase quantity of five as well.) The search.jsp page also contains a button that directs the user to cart.jsp.
- cart.jsp: Lists all the books currently in the shopping basket. The listing should include all items in the shopping basket with the product name, price, a text box for the quantity (which the user can use to change quantities of items), and a button to remove the item from the shopping basket. This page has three other buttons: one button to continue shopping (which returns the user to page index.jsp), a second button to update the shop-

ping basket with the altered quantities from the text boxes, and a third button to place the order, which directs the user to the page confirm.jsp.

- coni irm. j sp: Lists the complete order so far and allows the user to enter his or her contact information or customer ID. There are two buttons on this page: one button to cancel the order and a second button to submit the final order. The cancel button ernpties the shopping basket and returns the user to the home page. The submit button updates the database with the new order, empties the shopping basket, and returns the user to the home page.

DBDudes also considers the use of JavaScript at the presentation tier to check user input before it is sent to the middle tier. For example, in the page login. j sp, DBDudes is likely to write JavaScript code similar to that shown in Figure 7.12.

This leaves DBDudes with one final decision: how to connect applications to the DBMS. They consider the two main alternatives presented in Section 7.7: CGI scripts versus using an application server infrastructure. If they use CGI scripts, they would have to encode session management logic-not an easy task. If they use an application server, they can make use of all the functionality that the application server provides. Therefore, they recommend that B&N implement server-side processing using an application server.

B&N accepts the decision to use an application server, but decides that no code should be specific to any particular application server, since B&N does not want to lock itself into one vendor. DBDudes agrees proceeds to build the following pieces:

- DBDudes designs top level pages that allow customers to navigate the website as well as various search forms and result presentations.
- Assuming that DBDudes selects a Java-based application server, they have to write Java servlets to process form-generated requests. Potentially, they could reuse existing (possibly commercially available) JavaBeans. They can use JDBC as a database interface; exarnples of JDBC code can be found in Section 6.2. Instead of prograrnming servlets, they could resort to Java Server Pages and annotate pages with special .JSP markup tags.
- DBDudes select an application server that uses proprietary markup tags, but due to their arrangement with B&N, they are not allowed to use such tags in their code.

For completeness, we remark that if DBDudes and B&N had agreed to use CGr scripts, DBDucles would have had the following tasks:

- Create the top level HTML pages that allow users to navigate the site and various forms that allow users to search the catalog by ISBN, author name, or title. An example page containing a search form is shown in Figure 7.1. In addition to the input forms, DBDudes must develop appropriate presentations for the results.
- Develop the logic to track a customer session. Relevant information must be stored either at the server side or in the customer's browser using cookies.
- Write the scripts that process user requests. For example, a customer can use a form called 'Search books by title' to type in a title and search for books with that title. The CGI interface communicates with a script that processes the request. An example of such a script written in Perl using the DBI library for data access is shown in Figure 7.16.

Our discussion thus far covers only the customer interface, the part of the website that is exposed to B&N's customers. DBDudes also needs to add applications that allow the employees and the shop owner to query and access the database and to generate summary reports of business activities.

Complete files for the case study can be found on the webpage for this book.

7.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are URIs and URLs? (Section 7.2.1)
- How does the HTTP protocol work? What is a stateless protocol? (Section 7.2.2)
- Explain the main concepts of HTML. Why is it used only for data presentation and not data exchange? (Section 7.3)
- What are some shortcomings of HTML, and how does XML address them? (Section 7.4)
- What are the main components of an XML document? (Section 7.4.1)
- Why do we have XML DTDs? What is a well-formed XML document? What is a valid XML document? Give an example of an XML document that is valid but not well-formed, and vice versa. (Section 7.4.2)
- 'What is the role of domain-specific DTDs? (Section 7.4.3)
- What is a three-tier architecture? 'What advantages does it offer over single-tier and two-tier architectures? Give a short overview of the functionality at each of the three tiers. (Section 7.5)

- Explain how three-tier architectures address each of the following issues of database-backed Internet applications: heterogeneity, thin clients, data integration, scalability, software development. (Section 7.5.3)
- Write an HTML form. Describe all the components of an HTML form. (Section 7.6.1)
- What is the difference between the HTML GET and POST methods? How does URI encoding of an HTML form work? (Section 7.11)
- What is JavaScript used for? Write a JavaScipt function that checks whether an HTML form element contains a syntactically valid email address. (Section 7.6.2)
- What problem do style sheets address? What are the advantages of using style sheets? (Section 7.6.3)
- What are Cascading Style Sheets? Explain the components of Cascading Style Sheets. What is XSL and how it is different from CSS? (Sections 7.6.3 and 7.13)
- What is CGl and what problem does it address? (Section 7.7.1)
- What are application servers and how are they different from webservers? (Section 7.7.2)
- What are servlets? How do servlets handle data from HTML forms? Explain what happens during the lifetime of a servlet. (Section 7.7.3)
- What is the difference between servlets and JSP? When should we use servlets and when should we use JSP? (Section 7.7.4)
- Why do we need to maintain state at the middle tier? What are cookies? How does a browser handle cookies? How can we access the data in cookies from servlets? (Section 7.7.5)

EXERCISES

Exercise 7.1 Briefly answer the following questions:

1. Explain the following terms and describe what they are used for: HTML, URL, XML, Java, JSP, XSL, XSLT, servlet, cookie, HTTP, ess, DTD.
2. What is eGl? Why was eGl introduced? What are the disadvantages of an architecture using eel scripts?
3. \What is the difference between a webserver and an application server? What fUl1cionality do typical application servers provide?
4. When is an XML document well-formed? When is an XML document valid?

Exercise 7.2 Briefly answer the following questions about the HTTP protocol:

1. What is a communication protocol?
2. "What is the structure of an HTTP request message? What is the structure of an HTTP response message? Why do HTTP messages carry a version field?
3. What is a stateless protocol? Why was HTTP designed to be stateless?
4. Show the HTTP request message generated when you request the home page of this book (`http://www.cs.wisc.edu/~dbbook`). Show the HTTP response message that the server generates for that page.

Exercise 7.3 In this exercise, you are asked to write the functionality of a generic shopping basket: you will use this in several subsequent project exercises. Write a set of JSP pages that displays a shopping basket of items and allows users to add, remove, and change the quantity of items. To do this, use a cookie storage scheme that stores the following information:

- The UserId of the user who owns the shopping basket.
- The number of products stored in the shopping basket.
- A product id and a quantity for each product.

When manipulating cookies, remember to set the Expires property such that the cookie can persist for a session or indefinitely. Experiment with cookies using JSP and make sure you know how to retrieve, set values, and delete the cookie.

You need to create five JSP pages to make your prototype complete:

- Index Page (index.jsp): This is the main entry point. It has a link that directs the user to the Products page so they can start shopping.
- Products Page (products.jsp): Shows a listing of all products in the database with their descriptions and prices. This is the main page where the user fills out the shopping basket. Each listed product should have a button next to it, which adds it to the shopping basket. (If the item is already in the shopping basket, it increments the quantity by one.) There should also be a counter to show the total number of items currently in the shopping basket. Note that if a user has a quantity of five of a single item in the shopping basket, the counter should indicate a total quantity of five. The page also contains a button that directs the user to the Cart page.
- Cart Page (cart.jsp): Shows a listing of all items in the shopping basket cookie. The listing for each item should include the product name, price, a text box for the quantity (the user can change the quantity of items here), and a button to remove the item from the shopping basket. This page has three other buttons: one button to continue shopping (which returns the user to the Products page), a second button to update the cookie with the altered quantities from the text boxes, and a third button to place or confirm the order, which directs the user to the Confirm page.
- Confirm Page (confirm.jsp): Lists the final order. There are two buttons on this page. One button cancels the order and the other submits the completed order. The cancel button just deletes the cookie and returns the user to the Index page. The submit button updates the database with the new order, deletes the cookie, and returns the user to the Index page.

Exercise 7.4 In the previous exercise, replace the page products.jsp with the following *search page* search.jsp. This page allows users to search products by name or description. There should be both a text box for the search text and radio buttons to allow the

user to choose between search-by-name and search-by-description (as well as a submit button to retrieve the results). The page that handles search results should be modeled after products.jsp (as described in the previous exercise) and be called products.jsp. It should retrieve all records where the search text is a substring of the name or description (as chosen by the user). To integrate this with the previous exercise, simply replace all the links to products.jsp with search.jsp.

Exercise 7.5 'Write a simple authentication mechanism (without using encrypted transfer of passwords, for simplicity). We say a user is authenticated if she has provided a valid username-password combination to the system; otherwise, we say the user is not authenticated. Assume for simplicity that you have a database schema that stores only a customer id and a password:

Passwords(cid: integer, username: string, password: string)

1. How and where are you going to track when a user is 'logged on' to the system?
2. Design a page that allows a registered user to log on to the system.
3. Design a page header that checks whether the user visiting this page is logged in.

Exercise 7.6 (Due to Jeff Derstadt) TechnoBooks.com is in the process of reorganizing its website. A major issue is how to efficiently handle a large number of search results. In a human interaction study, it found that modern users typically like to view 20 search results at a time, and it would like to program this logic into the system. Queries that return batches of sorted results are called *top N queries*. (See Section 25.5 for a discussion of database support for top N queries.) For example, results 1-20 are returned, then results 21-40, then 41-60, and so on. Different techniques are used for performing top N queries and TechnoBooks.com would like you to implement two of them.

Infrastructure: Create a database with a table called Books and populate it with some books, using the format that follows. This gives you 111 books in your database with a title of AAA, BBB, CCC, DDD, or EEE, but the keys are not sequential for books with the same title.

Books(*bookid:* INTEGER, *title:* CHAR(80), *author:* CHAR(80), *price:* REAL)

```
For i = 1 to 111 {
        Insert the tuple (i, "AAA", "AAA Author", 5.99)
        i=i+1
        Insert the tuple (i, "BBB", "BBB Author", 5.99)
        i = i + 1
        Insert the tuple (i, "CCC", "CCC Author", 5.99)
        i=i+1
        Insert the tuple (i, "DDD", "DDD Author", 5.99)
        1=i+1
        Insert the tuple (i, "EEE", "EEE Author", 5.99)
```

Placeholder Technique: The simplest approach to top N queries is to store a placeholder for the first and last result tuples, and then perform the same query. When the new query results are returned, you can iterate to the placeholders and return the previous or next 20 results.

Tuples Shown	Lower Placeholder	Previous Set	Upper Placeholder	Next Set
1-20	1	None	20	21-40
21-40	21	1-20	40	41-60
41-60	41	21-40	60	61-80

Write a webpage in JSP that displays the contents of the Books table, sorted by the Title and BookId, and showing the results 20 at a time. There should be a link (where appropriate) to get the previous 20 results or the next 20 results. To do this, you can encode the placeholders in the Previous or Next Links as follows. Assume that you are displaying records 21-40. Then the previous link is display.jsp?lower=21 and the next link is display.jsp?upper=40.

You should not display a previous link when there are no previous results; nor should you show a Next link if there are no more results. When your page is called again to get another batch of results, you can perform the same query to get all the records, iterate through the result set until you are at the proper starting point, then display 20 more results.

What are the advantages and disadvantages of this technique?

Query Constraints Technique: A second technique for performing top N queries is to push boundary constraints into the query (in the WHERE clause) so that the query returns only results that have not yet been displayed. Although this changes the query, fewer results are returned and it saves the cost of iterating up to the boundary. For example, consider the following table, sorted by (title, primary key).

Batch	Result Number	Title	Primary Key
1	1	AAA	105
1	2	BBB	13
1	3	eee	48
1	4	DDD	52
1	5	DDD	101
2	6	DDD	121
2	7	EEE	19
2	8	EEE	68
2	9	FFF	2
2	10	FFF	33
3	11	FFF	58
3	12	FFF	59
3	13	GGG	93
3	14	EHH	132
3	15	HHH	135

In batch 1, rows 1 through 5 are displayed, in batch 2 rows 6 through 10 are displayed, and so on. Using the placeholder technique, all 15 results would be returned for each batch. Using the constraint technique, batch 1 displays results 1-5 but returns results 1-15, batch 2 will display results 6-10 but returns only results 6-15, and batch 3 will display results 11-15 but return only results 11-15.

The constraint can be pushed into the query because of the sorting of this table. Consider the following query for batch 2 (displaying results 6-10):

```
EXEC SQL SELECT B.Title
FROM     Books B
WHERE    (B.Title = 'DDD' AND B.BookId > 101) OR (B.Title > 'DDD')
ORDER BY B.Title, B.Bookld
```

This query first selects all books with the title 'DDD,' but with a primary key that is greater than that of record 5 (record 5 has a primary key of 101). This returns record 6. Also, any book that has a title after 'DDD' alphabetically is returned. You can then display the first five results.

The following information needs to be retained to have Previous and Next buttons that return more results:

- Previous: The title of the *first* record in the previous set, and the primary key of the *first* record in the previous set.
- Next: The title of the *first* record in the next set; the primary key of the *first* record in the next set.

These four pieces of information can be encoded into the Previous and Next buttons as in the previous part. Using your database table from the first part, write a JavaServer Page that displays the book information 20 records at a time. The page should include *Previous* and *Next* buttons to show the previous or next record set if there is one. Use the constraint query to get the Previous and Next record sets.

PROJECT-BASED EXERCISES

In this chapter, you continue the exercises from the previous chapter and create the parts of the application that reside at the middle tier and at the presentation tier. More information about these exercises and material for more exercises can be found online at

```
http://www.cs.wisc.edu/~dbbook
```

Exercise 7.7 Recall the Notown Records website that you worked on in Exercise 6.6. Next, you are asked to develop the actual pages for the Notown Records website. Design the part of the website that involves the presentation tier and the middle tier, and integrate the code that you wrote in Exercise 6.6 to access the database.

I. Describe in detail the set of webpages that users can access. Keep the following issues in mind:

   - All users start at a common page.
   - For each action, what input does the user provide? How will the user provide it -by clicking on a link or through an HTML form?
   - What sequence of steps does a user go through to purchase a record? Describe the high-level application flow by showing how each IIser action is handled.

2. Write the webpages in HTML without dynamic content.
3. Write a page that allows users to log on to the site. Use cookies to store the information permanently at the user's browser.
4. Augment the log-on page with JavaScript code that checks that the username consists only of the characters from a to z.
5. Augment the pages that allow users to store items in a shopping basket with a condition that checks whether the user has logged on to the site. If the user has not yet logged on, there should be no way to add items to the shopping cart. Implement this functionality using JSP by checking cookie information from the user.
6. Create the remaining pages to finish the website.

Exercise **7.8** Recall the online pharmacy project that you worked on in Exercise 6.7 in Chapter 6. Follow the analogous steps from Exercise 7.7 to design the application logic and presentation layer and finish the website.

Exercise **7.9** Recall the university database project that you worked on in Exercise 6.8 in Chapter 6. Follow the analogous steps from Exercise 7.7 to design the application logic and presentation layer and finish the website.

Exercise **7.10** Recall the airline reservation project that you worked on in Exercise 6.9 in Chapter 6. Follow the analogous steps from Exercise 7.7 to design the application logic and presentation layer and finish the website.

BIBLIOGRAPHIC NOTES

The latest version of the standards mentioned in this chapter can be found at the website of the World Wide Web Consortium (www.w3.org). It contains links to information about I-ITML, cascading style sheets, XIvIL, XSL, and much more. The book by Hall is a general introduction to Web programming technologies [357]; a good starting point on the Web is www.Webdeve1oper.com. There are many introductory books on CGI prograruning, for example [210, 198]. The JavaSoft (java.sun.com) home page is a good starting point for Servlets, .JSP, and all other Java-related technologies. The book by Hunter [394] is a good introduction to Java Servlets. Microsoft supports Active Server Pages (ASP), a comparable tedmology to .lSI'. More information about ASP can be found on the Microsoft Developer's Network horne page (msdn.microsoft.com).

There are excellent websites devoted to the advancement of XML, for example www.xml.com and www.ibm.com/xm1. that also contain a plethora of links with information about the other standards. There are good introductory books on many diflerent aspects of XML, for example [195, 158,597,474, 381, 320]. Information about UNICODE can be found on its home page http://www.unicode.org.

Inforrnation about .lavaServer Pages ane! servlets can be found on the JavaSoft home page at java.sun.com at java.sun.com/products/jsp and at java.sun.com/products/servlet.