

MODULE 3

Chapter 1 SQL

The SQL language may be considered one of the major reasons for the commercial success of relational databases. The name SQL is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUery Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs.

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL and a DML. It also has rules for embedding SQL statements into a general- purpose programming language such as Java, COBOL, or C/C++.

3.1 SQL Data Definition and Data Types

SQL uses the terms table, row, and column for the formal relational model terms relation, tuple, and attribute, respectively. We will use the corresponding terms interchangeably. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers).

3.1.1 Schema and Catalog Concepts in SQL

The concept of an SQL schema was incorporated starting with SQL2 in order to group together tables and other constructs that belong to the same database application. An SQL schema is identified by a schema name, and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for each element in the schema. Schema elements include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema.

A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Alternatively, the schema can be assigned a name and authorization identifier, and the elements can be defined later. For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'. Note that each statement in SQL ends with a semicolon.

CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

3.1.2 The CREATE TABLE Command in SQL

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command. we can explicitly attach the schema name to the relation name, separated by a period.

For example, by writing

CREATE TABLE COMPANY.EMPLOYEE ...

rather than

CREATE TABLE EMPLOYEE ...

we can explicitly (rather than implicitly) make the EMPLOYEE table part of the COMPANY schema.

The relations declared through CREATE TABLE statements are called base tables (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS. Base relations are distinguished from virtual relations, created through the CREATE

VIEW statement which may or may not correspond to an actual physical file.

Example:

```
CREATE TABLE EMPLOYEE ( Fname VARCHAR(15) NOT NULL, Minit CHAR, Lname
VARCHAR(15) NOT NULL, Ssn CHAR(9) NOT NULL, Bdate DATE, Address
VARCHAR(30), Sex CHAR, Salary DECIMAL(10,2), Super_ssn CHAR(9), Dno INT NOT
NULL, PRIMARY KEY (Ssn), FOREIGN KEY (Super_ssn) REFERENCES
EMPLOYEE(Ssn), FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber)) ;
CREATE TABLE DEPARTMENT ( Dname VARCHAR(15) NOT NULL, Dnumber INT NOT
NULL, Mgr_ssn CHAR(9) NOT NULL, Mgr_start_date DATE, PRIMARY KEY (Dnumber),
UNIQUE (Dname), FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn));
```

3.1.3 Attribute Data Types and Domains in SQL

The basic data types available for attributes include numeric, character string, bit string, Boolean, date and time.

Numeric data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(i,j)—or DEC(i,j) or NUMERIC(i,j)—where i, the precision, is the total number of decimal digits and j, the scale, is the number of digits after the decimal point.

■ Character-string data types are either fixed length—CHAR(n) or CHARACTER(n), where n is the number of characters—or varying length— VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is case sensitive (a distinction is made between uppercase and lowercase).³ For fixed length strings, a shorter string is padded with blank

characters to the right. For example, if the value ‘Smith’ is for an attribute of type CHAR(10), it is padded with five blank characters to become ‘Smith ’ if needed.

Bit-string data types are either of fixed length n—BIT(n)—or varying length—BIT VARYING(n), where n is the maximum number of bits. The default for n, the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B‘10101’.

■ A **Boolean data type** has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.

■ The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD.

The **TIME** data type has at least eight positions, with the components HOUR, MINUTE and SECOND in the form HH:MM:SS.

■ A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single quoted strings preceded by the keyword TIMESTAMP, with a blank space between data and time; for example, TIMESTAMP '2008-09-27 09:12:47.648302'.

■ Another data type related to DATE, TIME, and TIMESTAMP is the **INTERVAL** data type. This specifies an interval—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

It is possible to specify the data type of each attribute directly as in Figure 4.1; alternatively, a domain can be declared, and the domain name used with the attribute specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

```
CREATE DOMAIN SSN_TYPE AS CHAR(9);
```

We can use SSN_TYPE in place of CHAR(9) in the above create table for the attributes Ssn and Super_ssn of EMPLOYEE, Mgr_ssn of DEPARTMENT, Essn of WORKS_ON, and Essn of DEPENDENT.

3.2 Specifying Constraints in SQL

Now will describe the basic constraints that can be specified in SQL as part of table creation. These include key and referential integrity constraints, restrictions on attribute domains and NULLs and constraints on individual tuples within a relation.

3.2.1 Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the primary key of each relation, but it can be specified for any other attributes whose values are required not to be NULL.

It is also possible to define a default value for an attribute by appending the clause DEFAULT <value> to an attribute definition.

Another type of constraint can restrict attribute or domain values using the CHECK clause following an attribute or domain definition.

For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:
Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber < 21);

Ex:

```
CREATE TABLE EMPLOYEE ( Dno INT NOT NULL DEFAULT 1, CONSTRAINT EMPPK  
PRIMARY KEY (Ssn), CONSTRAINT EMPDEPTFK FOREIGN KEY(Dno) REFERENCES  
DEPARTMENT(Dnumber) ON DELETE SET DEFAULT ON UPDATE CASCADE);
```

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement. For example, we can write the following statement:

```
CREATE DOMAIN D_NUM AS INTEGER CHECK (D_NUM > 0 AND D_NUM < 21 );
```

We can then use the created domain D_NUM as the attribute type for all attributes that refer to department numbers In create table , such as Dnumber of DEPARTMENT, Dnum of PROJECT, Dno of EMPLOYEE, and so on.

3.2.2 Specifying Key and Referential Integrity Constraints

The PRIMARY KEY clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a single attribute, the clause can follow the attribute directly.

Ex: Dnumber INT PRIMARY KEY;

The UNIQUE clause specifies alternate (secondary) keys, as illustrated in the DEPARTMENT and PROJECT table declarations.

Ex: Dname VARCHAR(15) UNIQUE;

Referential integrity is specified via the FOREIGN KEY clause. A referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified. The default action that SQL takes for an integrity violation is to reject the update operation that will cause a violation, which is known as the RESTRICT option. However, the schema designer can specify an alternative action to be taken by attaching a referential triggered action clause to any foreign key constraint. The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE. We illustrate this with the examples shown in create table Here, the database designer chooses ON DELETE SET NULL and ON UPDATE CASCADE for the foreign key Super_ssn of EMPLOYEE. This means that if the tuple for a supervising employee is deleted, the value of Super_ssn is automatically set to NULL for all employee tuples that were referencing the deleted employee tuple. On the other hand, if the Ssn value for a supervising employee is updated (say, because it was entered incorrectly), the new value is cascaded to Super_ssn for all employee tuples referencing the updated employee tuple.

3.2.3 Giving Names to Constraints

A constraint may be given a constraint name, following the keyword CONSTRAINT. The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint. Giving names to constraints is optional.

Ex: CONSTRAINT EMPPK PRIMARY KEY (Ssn).

3.2.4 Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other table constraints can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called tuple-based constraints because they apply to each tuple individually and are checked whenever a tuple is inserted or modified. For example, suppose that the DEPARTMENT table in Figure 4.1 had an additional attribute Dept_create_date, which stores the date when the department was created. Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date.

```
CHECK (Dept_create_date <= Mgr_start_date);
```

3.3 schema change statements in sql

3.3.1 the drop command

The DROP command can be used to drop named schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is no longer needed, the DROP SCHEMA command can be used. 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.

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 8.1, 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 8.8). With the CASCADE option, all such constraints and views that reference the table are 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 the records, but leave the table definition for future use, then the DELETE command (see Section 8.6.2) 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.

3.3.2 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 relations in the COMPANY schema, 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 (see Section 8.6). 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 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 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 8.2 from the EMPLOYEE relation, we write:

ALTER TABLE COMPANY.EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;

Once this is done, we can redefined a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the ADD 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. There are many others details and options; we refer the interested reader to the SQL documents listed in the Selected Bibliography at the end of this chapter. The next two sections discuss the querying capabilities of SQL.

3.4 Basic Retrieval Queries in SQL

SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values. Hence, in general, an SQL table is not a set of tuples, because a set does not allow two identical members; rather, it is a multiset (sometimes called a bag) of tuples.

3.4.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries

The basic form of the SELECT statement, sometimes called a mapping or a select-from-where block, is formed of the three clauses SELECT, FROM, and WHERE and has the following form

Figure 5.

One possible database state for the COMPANY relational database schem

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

SELECT <attribute list>

FROM <table list>

WHERE <condition>;

where<attribute list> is a list of attribute names whose values are to be retrieved by the query.

<table list> is a list of the relation names required to process the query.

<condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

MODULE 3

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>(not equal operator).

Query 1.

Ex: Retrieve the birth date and address of the employee(s) whose name is 'John B.Smith'.

SELECT Bdate, Address FROM EMPLOYEE WHERE Fname='John' AND Minit='B' AND Lname='Smith';

(a)

<u>Bdate</u>	<u>Address</u>
1965-01-09	731 Fondren, Houston, TX

Query 1.a.

Retrieve the name and address of all employees who work for the 'Research' department.

SELECT Fname, Lname, Address FROM EMPLOYEE, DEPARTMENT WHERE Dname='Research' AND Dnumber=Dno;

(b)

<u>Fname</u>	<u>Lname</u>	<u>Address</u>
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

In the WHERE clause of query, the condition Dname = 'Research' is a selection condition that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a join condition, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE.

A query that involves only selection and join conditions plus projection attributes is known as a select-project-join query. The next example is a select-project-join query with two join conditions.

Query 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

SELECT Pnumber, Dnum, Lname, Address, Bdate FROM PROJECT, DEPARTMENT, EMPLOYEE WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Plocation='Stafford';

(c)

<u>Pnumber</u>	<u>Dnum</u>	<u>Lname</u>	<u>Address</u>	<u>Bdate</u>
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

The join condition Dnum = Dnumber relates a project tuple to its controlling department tuple, whereas the join condition Mgr_ssn=Ssn relates the controlling department tuple to the employee tuple who manages that department.

3.4.2 Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different relations. If this is the case and a multitable query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity. This is done by prefixing the relation name to the attribute name and separating the two by a period.

SELECT Fname, EMPLOYEE E.Name, Address FROM EMPLOYEE, DEPARTMENT WHERE DEPARTMENT.Name='Research' AND DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;

We can also create an alias for each table name to avoid repeated typing of long table names.

For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE AS E, EMPLOYEE AS S WHERE E.Super_ssn=S.Ssn;

(d)

<u>E.Fname</u>	<u>E.Lname</u>	<u>S.Fname</u>	<u>S.Lname</u>
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jennifer	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace

In this case, we are required to declare alternative relation names E and S, called aliases or tuple variables, for the EMPLOYEE relation. An alias can follow the keyword AS, as shown in Query, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of Query.

It is also possible to rename the relation attributes within the query in SQL by giving them aliases. For example, if we write EMPLOYEE AS E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Ssn, Dno) in the FROM clause, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on.

In Query, we can think of E and S as two different copies of the EMPLOYEE relation; the first, E, represents employees in the role of supervisees or subordinates; the second, S, represents employees in the role of supervisors.

3.4.3 Unspecified WHERE Clause and Use of the Asterisk

A missing WHERE clause indicates no condition on tuple selection; hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—all possible tuple combinations—of these relations is selected.

Queries 9 and 10.

Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

Q9: SELECT Ssn FROM EMPLOYEE;

123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

Q10: SELECT Ssn, Dname FROM EMPLOYEE, DEPARTMENT;

(f)

Ssn	Dname
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

It is extremely important to specify every selection and join condition in the WHERE clause; if any such condition is overlooked, incorrect and very large relations may result.

3.4.4 Tables as Sets in SQL

SQL usually treats a table not as a set but rather as a multiset; duplicate tuples can appear more than once in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries.

If we do want to eliminate duplicate tuples from the result of an SQL query, we use the keyword DISTINCT in the SELECT clause, meaning that only distinct tuples should remain in the result.

Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

Q11: SELECT ALL Salary FROM EMPLOYEE;

Q11A: SELECT DISTINCT Salary FROM EMPLOYEE;

In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not.

Figure 4.4
Results of additional SQL queries when applied to the COMPANY database state shown in Figure 3.6.
(a) Q11. (b) Q11A.

(a)	Salary	(b)	Salary
	30000		30000
	40000		40000
	25000		25000
	43000		43000
	38000		38000
	25000		55000
	25000		
	55000		

SQL has directly incorporated some of the set operations from mathematical set theory, which are also part of relational algebra (see Chapter 6). There are set union (UNION), set difference (EXCEPT) and set intersection (INTERSECT) operations. The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result. These set operations apply only to union-compatible relations, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations. The next example illustrates the use of UNION.

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

(SELECT DISTINCT Pnumber FROM PROJECT, DEPARTMENT, EMPLOYEE WHERE Dnum=Dnumber AND Mgr_ssn=Ssn AND Lname='Smith') UNION

(SELECT DISTINCT Pnumber FROM PROJECT, WORKS_ON, EMPLOYEE WHERE Pnumber=Pno AND Essn=Ssn AND Lname='Smith');

(a) R S

R	S
A	A
a1	a1
a2	a2
a2	a4
a3	a5

(b) T

T
A
a1
a1
a2
a2
a2
a3
a4
a5

(c) T

T
A
a2
a3

(d) T

T
A
a1
a2

Figure 4.5

The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

3.4.5 Substring Pattern Matching and Arithmetic Operators

In this section we discuss several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character. For example, consider the following query.

Query 12. Retrieve all employees whose address is in Houston, Texas.

```
SELECT Fname, Lname FROM EMPLOYEE WHERE Address LIKE '%Houston,TX%';
```

Query 12A. Find all employees who were born during the 1950s.

```
SELECT Fname, Lname FROM EMPLOYEE WHERE Bdate LIKE '__ 5 ';
```

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (−), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains.

Query 13. Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

```
SELECT E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal FROM EMPLOYEE AS E,  
WORKS_ON AS W, PROJECT AS P WHERE E.Ssn=W.Essn AND W.Pno=P.Pnumber AND  
P.Pname='ProductX';
```

Another comparison operator, which can be used for convenience, is BETWEEN, which is illustrated in Query 14.

Query 14. Retrieve all employees in department 5 whose salary is between \$30,000 and \$40,000.

```
SELECT * FROM EMPLOYEE WHERE (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

The condition (Salary BETWEEN 30000 AND 40000) in Q14 is equivalent to the condition ((Salary >= 30000) AND (Salary <= 40000)).

3.4.6 Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the ORDER BY clause.

Ex: `SELECT Salary FROM EMPLOYEE ORDER BY Salary;`

The default order is in ascending order of values. We can specify the keyword DESC if we want to see the result in a descending order of values. The keyword ASC can be used to specify ascending order explicitly.

Discussion and Summary of Basic SQL Retrieval Queries

A simple retrieval query in SQL can consist of up to four clauses, but only the first two—SELECT and FROM—are mandatory. The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

`SELECT <attribute list> FROM <table list> [WHERE <condition> ] [ORDER BY <attribute list> ];`

The SELECT clause lists the attributes to be retrieved, and the FROM clause specifies all relations (tables) needed in the simple query. The WHERE clause identifies the conditions for selecting the tuples from these relations, including join conditions if needed. ORDER BY specifies an order for displaying the results of a query.

3.5 INSERT, DELETE, and UPDATE Statements in SQL

3.5.1 The INSERT Command

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.

Ex: `INSERT INTO EMPLOYEE VALUES ( 'Richard','K','Marini','653298653','1962-12-30','98 Oak Forest, Katy, TX','M', 37000,'653298653', 4 );`

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many

attributes but only a few of those attributes are assigned values in the new tuple.

```
INSERT INTO EMPLOYEE (Fname, Lname, Dno, Ssn) VALUES ('Richard','Marini',  
4,'653298653');
```

Attributes not specified in the above query are set to their DEFAULT or to NULL, and the values are listed in the same order as the attributes are listed in the INSERT command itself.

A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the result of a query.

```
CREATE TABLE WORKS_ON_INFO (Emp_name VARCHAR(15), Proj_name  
VARCHAR(15), Hours_per_week DECIMAL(3,1) );
```

```
INSERT INTO WORKS_ON_INFO (Emp_name, Proj_name, Hours_per_week )
```

```
SELECT E.Lname, P.Pname, W.Hours FROM PROJECT P, WORKS_ON W, EMPLOYEE E  
WHERE P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

3.5.2 The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if referential triggered actions are specified in the referential integrity constraints of the DDL. Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table.

```
Ex: DELETE FROM EMPLOYEE WHERE Lname='Brown'; DELETE FROM EMPLOYEE  
WHERE Ssn='123456789'; DELETE FROM EMPLOYEE WHERE Dno=5;  
DELETE FROM EMPLOYEE;
```

The above DELETE commands if applied independently to the database in Figure 3.6, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

3.5.3 The UPDATE Command

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL. An additional SET clause in the UPDATE command specifies the attributes to be modified and their new values.

For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use the below query

```
UPDATE PROJECT SET Plocation = 'Bellaire', Dnum =5 WHERE Pnumber=10;
```

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary.

```
UPDATE EMPLOYEE SET Salary = Salary * 1.1 WHERE Dno = 5;
```

It is also possible to specify NULL or DEFAULT as the new attribute value.

3.6 Additional Features of SQL

SQL has various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification.

SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Persistent Stored Modules).

Each commercial RDBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes.

SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes.

SQL has language constructs for specifying the granting and revoking of privileges to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement—such as SELECT, INSERT, DELETE, or UPDATE—to access the relation.

In addition, the DBA staff can grant the privileges to create schemas, tables, or views to certain users. These SQL commands—called GRANT and REVOKE—

SQL has language constructs for creating triggers. These are generally referred to as active database techniques, since they specify actions that are automatically triggered by events such as database updates.

SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as object-relational.

SQL and relational databases can interact with new technologies such as XML.

CHAPTER 2 SQL: Advanced Queries

More Complex SQL Retrieval Queries

2.1 Comparisons Involving NULL and Three-Valued Logic

- SQL has various rules for dealing with NULL values.
- NULL is used to represent a missing value, but that it usually has one of three different interpretations:
 - value unknown (exists but is not known),
 - value not available (exists but is purposely withheld),
 - or value not applicable (the attribute is undefined for this tuple).

Consider the following examples to illustrate each of the meanings of NULL.

- **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database.
- **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.
- **Not applicable attribute.** An attribute Last College Degree 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 between the different meanings of NULL.
- When a NULL 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 5.1 shows the resulting values.

Table 5.1 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 5.1(a) and 5.1(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.
- For example, the result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN.
- 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.
- Query 18. Retrieve the names of all employees who do not have supervisors.

```
SELECT Fname, Lname
```

```
FROM EMPLOYEE WHERE Super_ssn IS NULL;
```

2.2 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 the WHERE clause of another query. That other query is called the outer query.
- IN operator
- Which is a comparison operator that 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.

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

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

Ex: 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');
```

- In this example, the IN operator compares the sub tuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.
- 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.

Ex: 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);
```

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.Fame, E.Lname FROM EMPLOYEE AS E
WHERE E.Ssn IN ( SELECT Essn FROM DEPENDENT AS D
WHERE E.Fame=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.

2.3 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.

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

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

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:

```
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;
```

2.4 The EXISTS and UNIQUE Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated 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 EXIST 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);
```

In general, EXISTS (Q) returns TRUE if there is at least one tuple in the result of the nested query Q, and it 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 it returns FALSE otherwise. Next, we illustrate the use of NOT EXISTS.

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

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

2.5 Explicit Sets and Renaming of Attributes in SQL

Explicit Sets

- 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.

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

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

Renaming of Attributes

- 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, and it can be used in both the SELECT and FROM clauses.

Ex. 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 in the query result.

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

2.6 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.

Ex. 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 first, 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 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.

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

There are a variety of outer join operations.

- **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 NULLvalues 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 NULLvalues for the attributes of the left table).
- **FULL OUTER JOIN:** It is a combination of left and right outer joins.
- The keyword **CROSS JOIN** is used to specify the **CARTESIAN PRODUCT** operation 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**.

EX: 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** using this syntax, we could write the query as follows:

Ex. SELECT E.Lname, S.Lname
 FROM EMPLOYEE E, EMPLOYEE S
 WHERE E.Super_ssn += S.Ssn;

2.7 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.
- A number of built-in **aggregate** functions exist: **COUNT, SUM, MAX, MIN, and AVG.**
- 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.

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

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

Ex. 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.

```
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=NUMBER AND DNAME = ‘Research’;
```

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.

2.8 Grouping: The GROUP BY and HAVING Clauses

GROUP BY clause

- SQL has a GROUP BY clause. 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).
- 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 non overlapping 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.

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 grouping attribute Dno. Hence, each group contains the employees who work in the same department. The COUNT and AVG functions are grouping attribute and the aggregate functions to be applied on each group of tuples.

Figure 5.1 (a) illustrates how grouping works on Q24; it also shows the result of Q24.

Figure 5.1

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	Bong	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

HAVING clause

- SQL provides a **HAVING** clause, which can appear in conjunction with a **GROUP BY** clause. 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.

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 while selection conditions in the WHERE clause limit the tuples to which functions are applied, the HAVING clause serves to choose whole groups. Figure 5.1b illustrates the use of HAVING and displays the result of Q26.

(b)

Pname	<u>Pnumber</u>	...	<u>Essn</u>	<u>Pno</u>	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

After applying the WHERE clause but before applying HAVING

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

Pname	<u>Pnumber</u>	...	<u>Essn</u>	<u>Pno</u>	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)

2.9 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 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.

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.
- **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 early.
- **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.

Specifying General Constraints as Assertions in SQL

ASSERTIONS

- In SQL, users can specify general constraints—those that do not fall into any of the categories described via declarative assertions, using the **CREATE ASSERTION** statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the **WHERE** clause of an SQL query.

For example, to specify the constraint that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for in SQL. we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS (SELECT *
    ,
        FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
        WHERE E.Salary>M.Salary
        AND E.Dno=-D.Dnumber
        AND D.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 data- base state for the assertion to be satisfied. The constraint name can be used later to refer to the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any **WHERE** clause condition can be used, but many constraints can be specified using the **EXISTS** and **NOT EXISTS** style of SQL conditions. Whenever some tuples in the database cause the condition of an **ASSERTION** statement to evaluate to

FALSE, the constraint 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.

2.10 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. The CREATE TRIGGER statement is used to implement such actions in SQL.

A typical trigger has **three components**:

Event: When this event happens, the trigger is activated

Condition (optional): If the condition is true, the trigger executes, otherwise skipped

Action: The actions performed by the trigger

The **action** is to be executed **automatically** if the **condition** is satisfied when **event** occurs.

Trigger: Events

Three event types

- Insert
- Update
- Delete

Two triggering times

- Before the event
- After the event

Two granularities

- Execute for each row
- Execute for each statement

Create Trigger <name> **Before|After** **Insert|Update|Delete** **ON** <tablename>
....

Trigger name

That is the event

• Example

Create Trigger ABC
Before Insert On Students
....

This trigger is activated when an insert statement is issued, but before the new record is inserted

Create Trigger XYZ
After Update On Students
....

This trigger is activated when an update statement is issued and after the update is executed

Create Trigger <name>
Before| After **Insert| Update| Delete**
For Each Row | For Each Statement
....

This is the event

This is the granularity

Create Trigger XYZ
After Update ON <tablename>

For each statement
....

This trigger is activated once (per UPDATE statement) after all records are updated

Create Trigger XYZ
Before Delete ON <tablename>

For each row
....

This trigger is activated before deleting each record

Trigger: Condition

```
Create Trigger <name>
Before| After      Insert| Update| Delete On <tableName>
For Each Row | For Each Statement

When <condition> ← That is the condition
...
```

If the employee salary > 150,000 then some actions will be taken

```
Create Trigger EmpSal
After Insert or Update On Employee
For Each Row
When (new.salary > 150,000)
...
```

Trigger: Action

- The new values of inserted or updated records **(:new)**
- The old values of deleted or updated records **(:old)**

```
Trigger body {
Create Trigger EmpSal
After Insert or Update On Employee
For Each Row
When (new.salary > 150,000)
Begin
    if (:new.salary < 100,000) ...
End;
}
```

Inside "When", the "new" and "old" should not have "."

Inside the trigger body, they should have "."

Example 1

```
CREATE TRIGGER init_count BEFORE INSERT ON student      /* Event*/
DECLARE
    Count integer;
BEGIN
    Count =0;                                           /* Action*/
END
```

Example 2

```
CREATE TRIGGER incr_count AFTER INSERT ON student      /* Event*/
WHEN (new.age<18)                                     /*Condition*/
FOR EACH ROW
BEGIN
    Count = count + 1;                                /* Action*/
END
```

1.10 Views (virtual table) in SQL**Concept of a View in SQL**

- A view is a single table that is derived from one or more base tables or other views
- Views neither exist physically nor contain data itself, it depends on the base tables for its existence
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

Specification of Views in SQL Syntax:

```
CREATE VIEW view_name AS SELECT column_name(s) FROM table_name
WHERE condition
```

Example

```
CREATE VIEW WORKS_ON1
AS SELECT  Fname, Lname, Pname, Hours FROM
EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn=Essn AND Pno=Pnumber ;
```

Ex. Retrieve the Last name and First name of all employees who work on 'ProductX'

```
SELECT      Fname, Lname FROMWORKS_ON1
WHERE       Pname='ProductX';
```

- A view always shows **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 If we do not need a view any more, we can use the **DROP VIEW** command `DROP VIEW WORKS_ON1;`

View Implementation and View Update View Implementation

- The problem of efficiently implementing a view for querying is complex Two main approaches have been suggested

Query Modification

- Modifying the view query into a query on the underlying base tables

Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are applied to the view within a short period of time.

Example

The query example# would be automatically modified to the following query by the DBMS

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

View Materialization

- Physically create a temporary view table when the view is first queried
- Keep that table on the assumption that other queries on the view will follow
- Requires efficient strategy for automatically updating the view table when the base tables are updated, that is **Incremental Update**
- **Incremental Update** determines what new tuples must be inserted, deleted, or modified in a materialized view table when a change is applied to one of the defining base table

View Update

- Updating of views is complicated and can be ambiguous
- An update on view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions.
- View involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways.

Example

Update the Pname attribute of 'john smith' from 'ProductX' to 'ProductY' 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 effect on the view. Two possible updates (a) and (b) on the base relations corresponding to above query .

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

- Update (a) relates 'john smith' to the 'ProductY' PROJECT tuple in place of the 'ProductX' PROJECT tuple and is the most likely update.
- Update (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'

OBSERVATIONS ON VIEWS

- 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 must be added at the end of the view definition if a view is to be updated.

Advantages of Views

- Data independence
- Currency
- Improved security
- Reduced complexity
- Convenience
- Customization
- Data integrity

Chapter 3

Database Application Development

Applications that rely on the DBMS to manage data run as separate processes that connect to the DBMS 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, but 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.

3.1 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.

3.1.1 Embedded SQL

Conceptually, embedding SQL commands in a host language program is straight forward. 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 from SQL commands.

The second complication has to do with SQL being set-oriented, and is addressed using cursors.

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 cage (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 SQL-92 standard defines a correspondence between the host language types and SQL types for a number of host languages. In our example, c_sname has the type CHARACTER (20) when referred to in an SQL statement, csid has the type INTEGER, crating has the type SMALLINT, and cage has the type REAL.

The SQL-92 standard recognizes two special variables for reporting errors, SQLCODE and SQLSTATE.

Embedding SQL Statements

All SQL statements 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.

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, :csid, :crating, :cage);
```

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 IF NOT FOUND] [ CONTINUE | GOTO stmt]
```

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.

3.1.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an impedance mismatch occurs because SQL operates on set 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:
- We usually need to open a cursor if the embedded statement is a SELECT query. However, we can avoid opening a cursor if the answer contains a single row.
- INSERT, DELETE, and UPDATE statements 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 command 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.

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 READONLY I FOR UPDATE]
```

cursor can be declared to be 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 an updatable simple variants of the UPDATE and DELETE commands allow us to update or delete the row on which the cursor is positioned. 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; If the keyword INSENSITIVE is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. A holdable cursor is specified using the WITH HOLD clause, and is not closed when the transaction is committed. 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!

3.1.3 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 SOL 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.

3.2 AN INTRODUCTION TO JDBC

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).

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.

3.2.1 JDBC Architecture

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

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 establishes the connection with the data source. The data source processes commands from the driver and returns the results.

Drivers in JDBC are classified 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 application 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 network to a middleware server that translates the JDBC requests into DBMS-specific method invocations. In this case the driver on the client site (Ie., 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 I 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.

3.3 JDBC CLASSES AND INTERFACES

JDBC is a collection of Java classes and interfaces that enables database access from programs 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;

3.3.1 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");
```

3.3.2 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 3.2 establishes a connection to an Oracle database assuming

that the strings `userId` and `password` are set to valid values.

```
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 3.2 Establishing a Connection with DId.

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.

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.

3.3.3 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, Prepared Statement, and Callable Statement. The Statement class is the base class for the other two statement classes. It allows us to query the data source with any static or dynamically generated SQL query.

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

Consider the sample code using a Prepared Statement 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 "?" 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 to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use clear Parameters () before setting parameter values in order to remove any old data.

```
// 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

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 `"l"` 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`.

3.3.4 ResultSets

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 otherwise. 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.next() moves to the first record
// rs.next() 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

JDBC provides special data types and specifies their relationship to corresponding SQL data types. Figure 6.5 shows the access or methods in a ResultSet object for the most common SQL data types. With these access or methods, we can retrieve values from the current row of the query result referenced by the ResultSet object. There are two forms for each access or 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 access or methods.

```
ResultSet rs=stmt.executeQuery(sqIQuery);
String sqlQueryi
ResultSet rs = stmt.executeQuery(sqIQuery)
while (rs.nextO) {
    isbn = rs.getString(1);
    title = rs.getString(" TITLE");
    // process isbn and title
}
```

3.3.5 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() 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 () returns an SQLState identifier based on the SQL:1999 specification.
- Public int getErrorCode () retrieves a vendor-specific error code.
- Public SQLException getNextException() gets the next exception in a chain of exceptions associated with the current SQLException object.

An SQL Warning is a subclass of SQLException. Warnings are not severe as errors and the program can usually proceed without special handling of warnings. Warnings are not thrown like other exceptions, and they are not caught as part of the try-catch block around a java.sql statement. We need to specify test whether warnings exist. Connection, Statement, and ResultSet objects all have a getWarnings() method with which we can retrieve SQL warnings if they exist. 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.getNextWarning(); //get next warning  
    } con.clearWarnings();  
  
    stmt.executeUpdate( queryString );  
    warning = stmt.getWarnings();  
    while (warning != null) {
```

```

        // handleSQLWarnings           //code to process warning
        warning = warning.getNextWarningO;    //get next warning
    }
} // end try
catch (TYSQLEException SQLe) {
    // code to handle exception
} // end catch

```

Figure 6.6 Processing JDBC Warnings and Exceptions

3.3.6 Examining Database Metadata

we can use the Database MetaData object to obtain information about the database system itself, as well as information from 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. we list some methods here:

publicResultSet 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.

1) Public **int getMaxConnections0** throws SQLException. This function returns the maximum 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

3.4 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 title 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.

3.4.1 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.title + ", " + books.price0);
} books.close();
```

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

```
PreparedStatement stmt = connection.prepareStatement(
    "SELECT title, price FROM Books WHERE author = ?");
// set the parameter in the query and execute it
stmt.setString(1, author);
ResultSet rs = 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.title()`. 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 construct, similar to Embedded SQL. Both iterator types have the same performance; which iterator to use depends on the programmer's taste.

3.5 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. 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 easier. In addition, application programmers do not need to know the database schema if we encapsulate all database access into stored procedures. Although they are called stored procedures, they do not have to be procedures in a programming language sense; they can be functions.

3.5.1 Creating a Simple Stored Procedure

Let us look at the example stored procedure written in SQL shown in Figure (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;
```

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(10),  
    IN addedQty INTEGER)  
  
UPDATE Books  
  
SET   qty in stock = qtyjn stock + addedQty  
WHERE bookisbn = isbn
```

Figure 6.9 A Stored Procedure with Argument

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 client:

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

Figure 6.10 A Stored Procedure in Java

3.5.2 Calling Stored Procedures

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

CALL storedProcedureName(argument1, 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[/O];
long qty;
EXEC SOL END DECLARE SECTION

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

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatement class. CallableStatement is a subclass of PreparedStatement and provides the same functionality. A stored procedure could contain multiple SQL statements 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.executeQuery()
while (rs.next())
```

Calling Stored Procedures from SQLJ

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

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

// create the cursor
CustomerInfo customerinfo;

// call the stored procedure
#sql customerinfo = (CALL ShowNumberOfOrders);
while (customerinfo.nextO) {
```

```
System.out.println(customerinfo.cid0 + "," +  
customerinfo.count());  
  
}
```

3.5.3 SQL/PSM

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 (parameter1,..., parameterN)  
RETURNS sqlDataType  
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 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 O WHERE O.tid = custId); IF (numOrders>10)
```

```
THEN rating=2;
ELSEIF (numOrders>5) THEN
  rating=1; ELSE rating=0;
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
statements:
END LOOP
```

3.6 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(10), cid: INTEGER, cardnum: CHAR(16), 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:

```
8CREATE 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 Oreders 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=0; i<orderList.length(); i++)
    // now instantiate the parameters with values
    Order currentOrder = orderList[i];
    pstmt.clearParameters(); pstmt.setInt(1, ordernum);
    pstmt.setString(2, Order.getIsbnO);
    pstmt.setInt(3, dd);
    pstmt.setString(4, creditCardNum); pstmt.setInt(5, Order.getQty0);
    pstmt.setDate(6, null);
    pstmt.executeUpdate();
    } con.commit;
    catch (SQLException e){
    con.rollback();
    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 qty_in_stock of the corresponding book in the Books table is greater than zero. Otherwise, the stored procedure sets the ship_date to two weeks.

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

Figure 6.12 Trigger to Update the Shipping Date of New Orders