Module – 03

Chapter: 01 – SQL

SQL

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), types, and domains, as well as other constructs such as views, assertions, and triggers.
- Because the SQL specification is very large, we give a description of the most important features.

Schema and Catalog Concepts in SQL

- Early versions of SQL did not include the concept of a relational database schema; all tables (relations) were considered part of the same schema.
- A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions.
- 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.
  Ex:-

CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';

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 possibly attribute constraints, such as NOT NULL.
- Alternatively, 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
```

as in Figure 6.1, 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 table and its rows are actually created and stored as a file by the DBMS.
- To deal with this type of problem, these constraints can be left out of the initial CREATE TABLE statement, and then added later using the ALTER TABLE statement .
- We displayed all the foreign keys in Figure 6.1 to show the complete COMPANY schema in one place.

```
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),
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) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                     NOT NULL,
          Dlocation             VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname                 VARCHAR(15)             NOT NULL,
          Pnumber               INT                     NOT NULL,
          Plocation             VARCHAR(15),
          Dnum                  INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn                  CHAR(9)                 NOT NULL,
          Pno                   INT                     NOT NULL,
          Hours                 DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn                  CHAR(9)                 NOT NULL,
          Dependent_name        VARCHAR(15)             NOT NULL,
          Sex                   CHAR,
          Bdate                 DATE,
          Relationship          VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Figure 6.1 SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 5.7.

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

- 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.
- The default for scale is zero, and the default for precision is implementation-defined.

Character-string data types

- 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).
- 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. Padded blanks are generally ignored when strings are compared.

Bit-string data types

- 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’.5 Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images.
- As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G).
- For example, BLOB(30G) specifies a maximum length of 30 gigabits.

Boolean data types

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.

DATE data types

- 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. Only valid dates and times should be allowed by the SQL implementation.
- This implies that months should be between 1 and 12 and days must be between 01 and 31; furthermore, a day should be a valid day for the corresponding month.
- If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

Timestamp data types

- 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 '2014-09-27 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.
- The format of DATE, TIME, and TIMESTAMP can be considered as a special type of string.
- For example, we can create a domain SSN_TYPE by the following statement:

CREATE DOMAIN SSN_TYPE AS CHAR(9);

- A domain can also have an optional default specification via a DEFAULT clause, as we discuss later for attributes.
- These can then be used either as data types for attributes, or as the basis for creating tables.

Specifying Constraints in SQL

This section describes 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 using the CHECK clause.

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.
- It is also possible to define a default value for an attribute by appending the clause DEFAULT <value> to an attribute definition.
- The default value is included in any new tuple if an explicit value is not provided for that attribute.
- Figure 6.2 illustrates an example of specifying a default manager for a new department and a default department for a new employee.
- If no default clause is specified, the default default value is NULL for attributes that do not have the NOT NULL constraint.

```
CREATE TABLE EMPLOYEE
    ( ... ,
      Dno          INT          NOT NULL       DEFAULT 1,
    CONSTRAINT EMPPK
      PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
      FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                  ON DELETE SET NULL          ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
      FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                  ON DELETE SET DEFAULT       ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
    ( ... ,
      Mgr_ssn CHAR(9)          NOT NULL       DEFAULT '888665555',
      ... ,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                  ON DELETE SET DEFAULT       ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
    ( ... ,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                  ON DELETE CASCADE           ON UPDATE CASCADE);
```

Figure 6.2
Example illustrating how default attribute values and referential integrity triggered actions are specified in SQL.

```
Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber < 21);
```

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);
```

Specifying Key and Referential Integrity Constraints

- Because keys and referential integrity constraints are very important, there are special clauses within the CREATE TABLE statement to specify them.

```
Dnumber INT PRIMARY KEY,
```

The **UNIQUE** clause specifies alternate (unique) keys, also known as candidate keys as illustrated in the DEPARTMENT and PROJECT table declarations in Figure 6.1. The **UNIQUE** clause can also be specified directly for a unique key if it is a single attribute, as in the following example:

```
Dname VARCHAR(15) UNIQUE,
```

- Referential integrity is specified via the FOREIGN KEY clause, as shown in Figure 6.1.
- The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE.
- In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE.
- The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing attribute for SET DEFAULT.
- The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples.

Giving Names to Constraints

- Figure 6.2 also illustrates how 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. It is also possible to temporarily defer a constraint until the end of a transaction.

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 row-based constraints because they apply to each row individually and are checked whenever a row is inserted or modified.
- For example, suppose that the DEPARTMENT table in Figure 6.1 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);

- The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL.

Basic Retrieval Queries in SQL

- SQL has one basic statement for retrieving information from a database: the SELECT statement.
- The SELECT statement is not the same as the SELECT operation of relational algebra.
- There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually.
- We will use example queries specified on the schema of Figure 5.5 and will refer to the sample database state shown in Figure 5.6 to show the results of some of these queries.
- In this section, we present the features of SQL for simple retrieval queries.
- Before proceeding, we must point out an important distinction between the practical SQL model and the formal relational model discussed in.
- SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values.

The SELECT-FROM-WHERE Structure of Basic SQL Queries

Queries in SQL can be very complex.

We will start with simple queries, and then progress to more complex ones in a step-by-step manner.

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:

```
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.

- In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>.
- These correspond to the relational algebra operators =, <, ≤, >, ≥, and ≠, respectively, and to the C/C++ programming language operators =, <, <=, >, >=, and !=.
- The main syntactic difference is the not equal operator. SQL has additional comparison operators that we will present gradually.

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:   SELECT   Bdate, Address
      FROM     EMPLOYEE
      WHERE    Fname = 'John' AND Minit = 'B' AND Lname = 'Smith';
```

- This query involves only the EMPLOYEE relation listed in the FROM clause.
- The query selects the individual EMPLOYEE tuples that satisfy the condition of the WHERE clause, then projects the result on the Bdate and Address attributes listed in the SELECT clause.

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1:   SELECT   Fname, Lname, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    Dname = 'Research' AND Dnumber = Dno;
```

- In the WHERE clause of Q1, 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 result of query Q1 is shown in Figure 6.3(b). In general, any number of selection and join conditions may be specified in a single SQL query.

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.

```
Q2:   SELECT   Pnumber, Dnum, Lname, Address, Bdate
      FROM     PROJECT, DEPARTMENT, EMPLOYEE
      WHERE    Dnum = Dnumber AND Mgr_ssn = Ssn AND
               Plocation = 'Stafford'
```

Figure 6.3
Results of SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q0. (b) Q1. (c) Q2. (d) Q8. (e) Q9. (f) Q10. (g) Q1C.

(a)

Bdate	Address
1965-01-09	731Fondren, Houston, TX

(b)

Fname	Lname	Address
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

(c)

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

(d)

E.Fname	E.Lname	S.Fname	S.Lname
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

(e)

E.Fname
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(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

(g)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
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

- 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.
- Each tuple in the result will be a combination of one project, one department (that controls the project), and one employee (that manages the department).
- The projection attributes are used to choose the attributes to be displayed from each combined tuple. The result of query Q2 is shown in Figure 6.3(c).

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 tables.
- 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.

```
Q1A:  SELECT   Fname, EMPLOYEE.Name, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.Name = 'Research' AND
               DEPARTMENT.Dnumber = EMPLOYEE.Dnumber;
```

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. Q1 can be rewritten as Q1′ below with fully qualified attribute names. We can also rename the table names to shorter names by creating an *alias* for each table name to avoid repeated typing of long table names (see Q8 below).

```
Q1′:  SELECT   EMPLOYEE.Fname, EMPLOYEE.LName,
               EMPLOYEE.Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.DName = 'Research' AND
               DEPARTMENT.Dnumber = EMPLOYEE.Dno;
```

The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice, as in the following example.

Query 8. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
Q8:   SELECT   E.Fname, E.Lname, S.Fname, S.Lname
      FROM     EMPLOYEE AS E, EMPLOYEE AS S
      WHERE    E.Super_ssn = S.Ssn;
```

- 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 Q8, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of Q8.

- 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, Sssn, Dno)in the FROM clause, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on.
- In Q8, 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.

```
Q1B:  SELECT   E.Fname, E.LName, E.Address
      FROM     EMPLOYEE AS E, DEPARTMENT AS D
      WHERE    D.DName = 'Research' AND D.Dnumber = E.Dno;
```

Unspecified WHERE Clause and Use of the Asterisk

- We discuss two more features of SQL here. 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;

Q10:  SELECT   Ssn, Dname
      FROM     EMPLOYEE, DEPARTMENT;
```

- 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.
- To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (*), which stands for all the attributes.
- The * can also be prefixed by the relation name or alias; for example, EMPLOYEE.* refers to all attributes of the EMPLOYEE table.
- Query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 6.3(g)), query Q1D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT.

- In which he or she works for every employee of the 'Research' department, and Q10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

```
Q1C:   SELECT   *
       FROM     EMPLOYEE
       WHERE    Dno = 5;

Q1D:   SELECT   *
       FROM     EMPLOYEE, DEPARTMENT
       WHERE    Dname = 'Research' AND Dno = Dnumber;

Q10A:  SELECT   *
       FROM     EMPLOYEE, DEPARTMENT;
```

Tables as Sets in SQL

- As we mentioned earlier, 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, for the following reasons:
  - Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
  - The user may want to see duplicate tuples in the result of a query.
  - When an aggregate function (see Section 7.1.7) is applied to tuples, in most cases we do not want to eliminate duplicates.

(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)

Salary
30000
40000
25000
43000
38000
55000

(c)

Fname	Lname

(d)

Fname	Lname
James	Borg

Figure 6.4
Results of additional SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q11. (b) Q11A. (c) Q16. (d) Q18.

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

- Specifying SELECT with neither ALL nor DISTINCT as in our previous examples is equivalent to SELECT ALL.
- For example, Q11 retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query, as shown in Figure 6.4(a).
- If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary.
- By using the keyword DISTINCT as in Q11A, we accomplish this, as shown in Figure 6.4(b).

Query 11. 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;
```

- SQL has directly incorporated some of the set operations from mathematical set theory, which are also part of relational algebra.
- There are set union (UNION), set difference (EXCEPT),11 and set intersection (INTERSECT) operations.

(a)

R
A
a1
a2
a2
a3

S
A
a1
a2
a4
a5

(b)

T
A
a1
a1
a2
a2
a2
a3
a4
a5

(c)

T
A
a2
a3

(d)

T
A
a1
a2

Figure 6.5
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).

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.

```
Q4A:    ( 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' );
```

- The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project.
- SQL also has corresponding multiset operations, which are followed by the keyword ALL (UNION ALL, EXCEPT ALL, INTERSECT ALL).
- Their results are multisets (duplicates are not eliminated).
- The behavior of these operations is illustrated by the examples in Figure 6.5. Basically, each tuple—whether it is a duplicate or not is considered as a different tuple when applying these operations.

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.

```
Q12:  SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    Address LIKE '%Houston,TX%';
```

To retrieve all employees who were born during the 1970s, we can use Query Q12A. Here, '7' must be the third character of the string (according to our format for date), so we use the value '_ _ 5 _ _ _ _ _ _ _', with each underscore serving as a place-holder for an arbitrary character.

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

```
Q12:  SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    Bdate LIKE '_ _ 7 _ _ _ _ _ _ _';
```

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

```
Q13:  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';
```

- For string data types, the concatenate operator || can be used in a query to append two string values.
- For date, time, timestamp, and interval data types, operators Include incrementing (+) or decrementing (−) a date, time, or timestamp by an interval.
- In addition, an interval value is the result of the difference between two date, time, or timestamp values.
- 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.

```
Q14:  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)).

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. This is illustrated by Query 15.

Query 15. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

```
Q15:  SELECT    D.Dname, E.Lname, E.Fname, P.Pname
      FROM      DEPARTMENT AS D, EMPLOYEE AS E, WORKS_ON AS W,
                PROJECT AS P
      WHERE     D.Dnumber = E.Dno AND E.Ssn = W.Essn AND W.Pno =
                P.Pnumber
      ORDER BY  D.Dname, E.Lname, E.Fname;
```

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. For example, if we want descending alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause of Q15 can be written as

```
ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC
```

INSERT, DELETE, and UPDATE Statements in SQL

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE. We discuss each of these in turn.

The INSERT Command

- In its simplest form, INSERT is used to add a single tuple (row) to a relation (table).
- 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.
- For example, to add a new tuple to the EMPLOYEE relation shown in Figure 5.5 and specified in the CREATE TABLE EMPLOYEE … command in Figure 6.1, we can use U1:

```
U1:   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.
- However, the values must include all attributes with NOT NULL specification and no default value.
- Attributes with NULL allowed or DEFAULTvalues are the ones that can be left out.
- For example, to enter a tuple for a new EMPLOYEE for whom we know only the Fname, Lname, Dno, and Ssn attributes, we can use U1A:

```
U1A:  INSERT INTO   EMPLOYEE (Fname, Lname, Dno, Ssn)
      VALUES        ('Richard', 'Marini', 4, '653298653');
```

- Attributes not specified in U1A 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.
- It is also possible to insert into a relation multiple tuples separated by commas in a single INSERT command.

```
U2:   INSERT INTO   EMPLOYEE (Fname, Lname, Ssn, Dno)
      VALUES        ('Robert', 'Hatcher', '980760540', 2);
      (U2 is rejected if referential integrity checking is provided by DBMS.)

U2A:  INSERT INTO   EMPLOYEE (Fname, Lname, Dno)
      VALUES        ('Robert', 'Hatcher', 5);
      (U2A is rejected if NOT NULL checking is provided by DBMS.)
```

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*. For example, to create a temporary table that has the employee last name, project name, and hours per week for each employee working on a project, we can write the statements in U3A and U3B:

```
U3A:  CREATE TABLE    WORKS_ON_INFO
      ( Emp_name      VARCHAR(15),
        Proj_name     VARCHAR(15),
        Hours_per_week DECIMAL(3,1) );

U3B:  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;
```

- A table WORKS_ON_INFO is created by U3A and is loaded with the joined information retrieved from the database by the query in U3B.

```
CREATE TABLE    D5EMPS LIKE EMPLOYEE
(SELECT         E.*
FROM            EMPLOYEE AS E
WHERE           E.Dno = 5) WITH DATA;
```

The clause WITH DATA specifies that the table will be created and loaded with the data specified in the query, although in some implementations it may be left out.

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.
- We must use the DROP TABLE command to remove the table definition.
- The DELETE commands in U4A to U4D, if applied independently to the database state shown in Figure 5.6, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

```
U4A:    DELETE FROM     EMPLOYEE
        WHERE           Lname = 'Brown';
U4B:    DELETE FROM     EMPLOYEE
        WHERE           Ssn = '123456789';
U4C:    DELETE FROM     EMPLOYEE
        WHERE           Dno = 5;
U4D:    DELETE FROM     EMPLOYEE;
```

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 U5:

```
U5:   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% raise in salary, as shown in U6.
- In this request, the modified Salary value depends on the original Salary value in each tuple, so two references to the Salary attribute are needed.
- In the SET clause, the reference to the Salary attribute on the right refers to the old Salary value before modification, and the one on the left refers to the new Salary value after modification.

```
U6:   UPDATE   EMPLOYEE
      SET      Salary = Salary * 1.1
      WHERE    Dno = 5;
```

Additional Features of SQL

SQL has a number of additional features that we have not described in this chapter

but that we discuss elsewhere in the book. These are as follows:

- SQL features: various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, case statements, 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.
- 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.
- 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 and OLAP/data warehouses.

Chapter – 02: Advances Quieries

More Complex SQL Retrieval Queries More Complex SQL Retrieval Queries

described some basic types of retrieval queries in SQL.

Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex retrievals from the database.

Comparisons Involving NULL and Three-Valued Logic

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

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

Table 7.1 Logical Connectives in Three-Valued Logic

(a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	OR	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	NOT			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

- Each expression result would have a value of TRUE, FALSE, or UNKNOWN.
- Tuple combinations that evaluate to FALSE or UNKNOWN are not selected.
- 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.
- Query 18 illustrates NULL comparison by retrieving any employees who do not have a supervisor.

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

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

Nested Queries, Tuples, and Set/Multiset Comparisons

- Some queries require that existing values in the database be fetched and then used in a comparison condition.
- Such queries can be conveniently formulated by using nested queries, which are complete select-from-where blocks within another SQL query.
- That other query is called the outer query.
- These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed.
- Query 4 is formulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A.

- In the outer query, we use the OR logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

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

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

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

- This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on.
- The keyword ALL can also be combined with each of these operators.
- An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

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

- In general, we can have several levels of nested queries.

- To illustrate the potential ambiguity of attribute names in nested queries, consider Query 16.

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

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

Correlated Nested Queries

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

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

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

The EXISTS and UNIQUE Functions in SQL

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

- First, we formulate Query 16 in an alternative form that uses EXISTS as in Q16B:

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

- EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query.
- In Q16B, the nested query references the Ssn, Fname, and Sex attributes of the EMPLOYEE relation from the outer query.

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

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

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

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

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

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

- It corresponds to the way we will write this query in tuple relation calculus.
- There is another SQL function, UNIQUE(Q), which returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE.
- This can be used to test whether the result of a nested query is a set (no duplicates) or a multiset (duplicates exist).

Explicit Sets and Renaming in SQL

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

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

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

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

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

Joined Tables in SQL and Outer Joins

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

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

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

- If the user requires that all employees be included, a different type of join called OUTER JOIN must be used explicitly.
- There are several variations of OUTER JOIN, as we shall see.

- In the SQL standard, this is handled by explicitly specifying the keyword OUTER JOIN in a joined table, as illustrated in Q8B:

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

- In SQL, the options available for specifying joined tables include INNER JOIN (only pairs of tuples that match the join condition are retrieved, same as JOIN).
- LEFT OUTER JOIN (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table).
- RIGHT OUTER JOIN (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table), and FULL OUTER JOIN.

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

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

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

Aggregate Functions in SQL

- Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary.
- Grouping is used to create subgroups of tuples before summarization.
- Grouping and aggregation are required in many database applications, and we will introduce their use in SQL through examples.
- A number of built-in aggregate functions exist: COUNT, SUM, MAX, MIN, and AVG.
- The COUNT function returns the number of tuples or values as specified in a query.
- The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values.

- These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later).
- The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a total ordering among one another.
- We illustrate the use of these functions with several queries.

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

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

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

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

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

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

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

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

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

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

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;

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

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

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

Grouping: The GROUP BY and HAVING Clauses

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

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

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

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

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

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

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

- Sometimes we want to retrieve the values of these functions only for groups that satisfy certain conditions.
- For example, suppose that we want to modify Query 25 so that only projects with more than two employees appear in the result.
- SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause, for this purpose.

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

(a)

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

(b)

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

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

After applying the WHERE clause but before applying HAVING

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

After applying the HAVING clause condition

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

Result of Q26
(Pnumber not shown)

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

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

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

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

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

- For example, suppose that we want to count the total number of employees whose salaries exceed Rs.40,000 in each department, but only for departments where more than five employees work.
- Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause.
- Suppose that we write the following incorrect query:

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

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

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

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

Other SQL Constructs: WITH and CASE

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

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

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

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

Recursive Queries in SQL

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

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

- The view is initially empty. It is first loaded with the first level (supervisor, supervisee) Ssn combinations via the first part (SELECT SupervisorSss, Ssn FROMEMPLOYEE), which is called the base query.
- At this point, the result of the recursive query is in the view SUP_EMP.

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.
- GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result.
- The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries.

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.
- These built-in constraints can be specified within the CREATE TABLE statement of SQL.
- introduce CREATE TRIGGER, which can be used to specify automatic actions that the database system will perform when certain events and conditions occur.
- This type of functionality is generally referred to as active databases.
- We only introduce the basics of triggers in this chapter, and present a more complete discussion of active databases.

Specifying General Constraints as Assertions in SQL

- In SQL, users can specify general constraints—those that do not fall into any of the categories described in Sections 6.1 and 6.2— via declarative assertions, using the CREATE ASSERTION statement.
- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- For example, to specify the constraint that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT     *
                     FROM       EMPLOYEE E, EMPLOYEE M,
                                DEPARTMENT D
                     WHERE      E.Salary>M.Salary
                                AND     E.Dno = D.Dnumber
                                AND     D.Mgr_ssn = M.Ssn ) );
```

Introduction to Triggers in SQL

- In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied.
- For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation.
- A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs.
- The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to monitor the database.
- Other actions may be specified, such as executing a specific stored procedure or triggering other updates. The CREATE TRIGGER statement is used to implement such actions in SQL.
- We discuss triggers in detail in Section 26.1 when we describe active databases. Here we just give a simple example of how triggers may be used.
- Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database.
- Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically.

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

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

The event(s):

- These are usually database update operations that are explicitly applied to the database.
- In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor.
- The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write
- more than one trigger to cover all possible cases.

- These events are specified after the keyword BEFORE in our example, which means that the trigger
- should be executed before the triggering operation is executed.
- An alternative is to use the keyword AFTER, which specifies that the trigger should be
- executed after the operation specified in the event is completed.

The condition that determines whether the rule action should be executed:

- Once the triggering event has occurred, an optional condition may be evaluated. If no condition is specified, the action will be executed once the event occurs.
- If a condition is specified, it is first evaluated, and only if it evaluates to true will the rule action be executed.
- The condition is specified in the WHEN clause of the trigger.

The action to be taken:

- The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.
- In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL.

We show how views are specified, and then we discuss the problem of updating views and how views can be implemented by the DBMS.

Concept of a View in SQL

- A view in SQL terminology is a single table that is derived from other tables.6 These other tables can be base tables or previously defined views.
- A view does not necessarily exist in physical form; it is considered to be a virtual table, in contrast to base tables, whose tuples are always physically stored in the database.
- This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.
- We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.
- For example, referring to the COMPANY database in Figure 5.5, we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE, WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins.

- Then we can issue queries on the view, which are specified as single table retrievals rather than as retrievals involving two joins on three tables.
- We call the EMPLOYEE, WORKS_ON, and PROJECT tables the defining tables of the view.

Specification of Views in SQL

- In SQL, the command to specify a view is CREATE VIEW.
- The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view.
- If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.
- The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 7.2 when applied to the database schema of Figure 5.5.

```
V1:   CREATE VIEW    WORKS_ON1
      AS SELECT      Fname, Lname, Pname, Hours
         FROM        EMPLOYEE, PROJECT, WORKS_ON
         WHERE       Ssn = Essn AND Pno = Pnumber;

V2:   CREATE VIEW    DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT      Dname, COUNT (*), SUM (Salary)
         FROM        DEPARTMENT, EMPLOYEE
         WHERE       Dnumber = Dno
         GROUP BY    Dname;
```

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

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

- In V1, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case, WORKS_ON1 inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON.

- View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.
- We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables.
- For example, to retrieve the last name and first name of all employees who work on the 'ProductX' project, we can utilize the WORKS_ON1 view and specify the query as in QV1:

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

- Views are also used as a security and authorization mechanism.
- A view is supposed to be always up-to-date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes.

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

```
V1A:    DROP VIEW    WORKS_ON1;
```

View Implementation, View Update, and Inline Views

- The problem of how a DBMS can efficiently implement a view for efficient querying is complex. Two main approaches have been suggested.
- One strategy, called query modification, involves modifying or transforming the view query (submitted by the user) into a query on the underlying base tables.
- For example, the query QV1 would be automatically modified to the following query by the DBMS:

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

- The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple view queries are going to be applied to the same view within a short period of time.
- This view update is shown in UV1:

```
UV1:    UPDATE WORKS_ON1
        SET     Pname = 'ProductY'
        WHERE   Lname = 'Smith' AND Fname = 'John'
                AND Pname = 'ProductX';
```

- This query can be mapped into several updates on the base relations to give the desired update effect on the view.
- In addition, some of these updates will create additional side effects that affect the result of other queries.
- For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

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

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

- Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' PROJECT tuple and is the most likely desired update.

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

In summary, we can make the following observations:

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

- In SQL, the clause WITH CHECK OPTION should be added at the end of the view definition if a view is to be updated by INSERT, DELETE, or UPDATE statements.
- This allows the system to reject operations that violate the SQL rules for view updates.
- The full set of SQL rules for when a view may be modified by the user are more complex than the rules stated earlier.
- It is also possible to define a view table in the FROM clause of an SQL query.
- This is known as an in-line view. In this case, the view is defined within the query itself.

Views as Authorization Mechanisms

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

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

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

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

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

Schema Change Statements in SQL

- In this section, we give an overview of the schema evolution commands available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements.

- This can be done while the database is operational and does not require recompilation of the database schema.
- Certain checks must be done by the DBMS to ensure that the changes do not affect the rest of the database and make it inconsistent.

The DROP Command

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

DROP SCHEMA COMPANY CASCADE;

- If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has no elements in it;
- otherwise, the DROP command will not be executed. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.
- If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command.
- For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 6.1, we can get rid of the DEPENDENT relation by issuing the following command:

DROP TABLE DEPENDENT CASCADE;

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

The ALTER Command

- The definition of a base table or of other named schema elements can be changed by using the ALTER command.
- For base tables, the possible alter table actions include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints.

- For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema (see Figure 6.1), we can use the command

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

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

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

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

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

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

- This is specified by using the ADD CONSTRAINT keyword in the ALTER TABLE statement followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed.
- The preceding subsections gave an overview of the schema evolution commands of SQL. It is also possible to create new tables and views within a database schema using the appropriate commands.

Chapter – 03: Application Development

Accessing Databases from Application

- The use of SQL commands within a host language program is called Embedded SQL.
- Details of Embed~ded SQL also depend on the host language.
- Although similar capabilities are supported for a variety of host languages, the syntax sometimes varies.

Embedded SQL

- Conceptually, embedding SQL commands in a host language program is straightforward.
- 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.
- First, the data types recognized by SQL may not be recognized by the host language and vice versa.
- This mismatch is typically addressed by casting data values appropriately before passing them to or frorn SQL commands.

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.
- The SQL-92standard defines such a correspondence between the host language types and SQL types for a number of host languages.
- For example. we can declare variables c-sname, c_sid, c_mt'ing, and cage.

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

Embedding SQL Statements

- All SQL staternents embedded within a host program must be clearly marked, with the details dependent on the host language; in C, SQL statements must be prefixed by EXEC SQL.
- An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.
- As a simple example, the following Embedded' SQL statement inserts a row, whose column values me based on the values of the host language variables contained in it, into the Sailors relation:

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

- Observe that a semicolon terminates the command, as per the convention for terminating statements in C.
- The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task:

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

- The intent is that the value of SQLSTATE should be checked after each Embedded SQL statement is executed.

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.

Basic Cursor Definition and Usage

- Cursors enable us to examine, in the host language program, a collection of JWS computed by an Embedded SQL statement:
- we usually need to open a cursor if the embedded statement is a SELECT (i.e.) a query). However, we can avoid opening a cursor if the answer contains a single row, as we see shortly.
- INSERT, DELETE, and UPDATE staternents typically require no cursor, although some variants of DELETE and UPDATE use a cursor.

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

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

- The value of cminmting in the SQL query associated with the cursor is the value of this variable when we open the cursor.

Properties of Cursors

The general form of a cursor declaration is:

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

- A cursor can be declared to be a read-only cursor (FOR READ ONLY) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (FOR UPDATE).
- If it is Updatable, simple variants of the UPDATE and DELETE commands allow us to update or delete the row on which the cursor is positioned.
- For example, if sinfa is an updatable cursor and open, we can execute the following statement:

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

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

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

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

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

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

- Many situations require the use of Dynamic SQL.
- Interactive and Embedded SQL commands can be prepared once at compile-time and then re-executecl as often as desired.

AN INTRODUCTION TO JDBC

- Embedded SQL enables the integration of SQL with a general-purpose programming language.
- ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language.
- Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an application programming interface (API).
- In addition, using ODBC or JDBC, an application can access not just one DBMS but several different ones simultaneously.
- ODBC and JDBC achieve portability at the level of the executable by introducing an extra level of indirection.
- A driver is a software program that translates the ODBC or JDBC calls into DBMS-specific calls.

Architecture

- The architecture of JDBC has four main components: the application, the driver manager, several data source specific driveTs, and the corresponding data SOUTces.
- The application initiates and terminates the connection with a data source.
- It sets transaction boundaries, submits SQL statements, and retrieves the results ----all through a well-defined interface as specified by the JDBC API.
- The primary goal of the dr-iver manager is to load JDBC drivers and pass JDBC

function calls from the application to the correct driver.

- The driver manager also handles JDBC initialization and information calls from the applications and can log all function calls.
- Depending on the relative location of the data source and the application, several architectural scenarios are possible.
- Drivers in JDBC are cla.ssified into four types depending on the architectural relationship between the application and the data source:

Type I Bridges:

- ❖ This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS.
- ❖ An example is a JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source.

Type II Direct Thanslation to the Native API via N on-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 signiicantly better than a JDBC-ODBC bridge.

Type III--Network Bridges:

- ❖ The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations.
- ❖ In this case, the driver on the client site (Le., the network bridge) is not DBMS-specific.

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.

JDBC CLASSES AND INTERFACES

- JDBC is a collection of Java classes and interfaces that enables database access from prograrl1s written in the Java language.

- It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling.
- The classes and interfaces are part of the java. sql package.
- The package javax. sql adds, among other things, the capability of connection pooling and the Row-Set interface.
- JDBC 2.0 also includes the j avax. sql package, the JDBC Optional Package.
- The package javax. sql adds, among other things, the capability of connection pooling and the Row-Set interface.

JDBC Driver Management

- In JDBC, data source drivers are managed by the Driver manager class, which maintains a list of all currently loaded drivers.
- The Driver manager class has hods register Driver, deregister Driver, and get Drivers 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 for Name in the Class returns the Java class as specified in the argument string and executes its static constructor.

The following Java example code explicitly loads a JDBC driver:

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

- There are two other ways of registering a driver.
- We can include the driver with-Djdbc. drivers=oracle/jdbc. driver at the command line when we start the Java application.

Connections

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

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

- The code example shown in Figure 6.2 establishes a connection to an Oracle database assuming that the strings user id and password are set to valid values.
- In JDBC, connections can have different properties.

- If auto commit is set for a connection, then each SQL statement is considered to be its own transaction.
- The Connection class has methods to set the

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

Figure 6.2 Establishing a Connection with JDBC

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

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

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

Executing SQL Statements

- The execute Query method is used if the SQL statement returns data, such as in a regular SELECT query.
- The Statement class is the base class for the other two statment classes.
- It allows us to query the data source with any static or dynamically generated SQL query.
- JDBC supports three different ways of executing statements: Statement, PreparedStatement, and CallableStatement.
- It allows us to query the data source with any static or dynamically generated SQL query.
- The SQL query specifies the query string, but uses "1' for the values of the parameters, which are set later using methods setString, setFloat, and setlnt.
- Examples of places where they can appear include the WHERE clause (e.g., 'WHERE author=?'), or in SQL UPDATE and INSERT staternents, as in Figure 6.3.

ResultSets

- 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 0 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.nextO moves to the first record
// rs.nextO moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next()) {
    // process the data
}
```

Figure 6.4 Using a ResultSet Object

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

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

Matching Java and SQL Data Types

- In considering the interaction of an application with a data source, the issues we encountered in the context of Embedded SQL arise again.
- To deal with such issues, JDBC provides special data types and specifies their relationship to corresponding SQL data types.
- There are two forms for each accessor method: One method retrieves values by column index, starting at one, and the other retrieves values by column name.
- The following example shows how to access fields of the current ResultSet row using accessor methods.
- Figure 6.5 shows the accessor methods in a ResultSet object for the most common SQL datatypes.

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

Figure 6.5 Reading SQL Datatypes from a ResultSet Object

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

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.
- In addition to the standard getMessage 0 method inherited from Throwable, SQLException has two additional methods that provide further information, and a method to get (or chain) additional exceptions:

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

- An SQL Warning is a subclass of SQLException. Warnings are not AS severe as errors and the program can usually proceed without special handling of warnings.

- Connection, Statement, and ResultSet objects all have a get Warnings 0 method with which we can retrieve SQL warnings if they exist.
- Duplicate retrieval of warnings can be avoided through clear Warnings O.
- Warnings are not thrown like other exceptions, and they are not caught as part of the try"-catch block around a java.sql statement.
- Statement objects clear warnings automatically on execution of the next statement; ResultSet objects clear warnings every time a new tuple is accessed.

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

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

Figure 6.6 Processing JDBC Warnings and Exceptions

Examining Database Metadata

we can use the Database MetaData object to obtain information about the database system itself, as well as information frorn the database catalog.

For example, the following code fragment shows how to obtain the name and driver version of the JDBC driver:

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

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

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

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

- public ResultSet getCatalogs 0 throws SqLException. This function returns a ResultSet that can be used to iterate over all the catalog relations. The functions getIndexInfo 0 and getTables 0 work analogously.
- pUblic int getMaxConnections 0 throws SqLException. This function returns the ma.ximum number of connections possible.

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

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

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

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

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, the following SQLJ statement binds host language variables title, price, and author to the return values of the cursor books.

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

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

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

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

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

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 :titIe, :price
    FROM Books WHERE author = :author
};

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

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

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

// set the parameter in the query ancl execute it
stmt.setString(1, author);
ResultSet 18 = 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.

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.
- Stored procedures are also beneficial for software engineering reasons.

- Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easy.
- In addition, application programmers do not need to know the database schema if we encapsulate all databa.'3e access into stored procedures.

Creating a Simple Stored Procedure

- Let us look at the example stored procedure written in SQL shown in Figure 6.8.
- we see that stored procedures must have a name; this stored procedure has the name 'ShowNumberOfOrders.' Otherwise, it just contains an SQL statement that is precompiled and stored at the server.

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

Figure 6.8 A Stored Procedure in SQL

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

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

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

Figure 6.9 A Stored Procedure with Arguments

Calling Stored Procedures

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

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

Figure 6.10 A Stored Procedure in Java

CALL stored ProcedureName (argumentl, argument2, ... , argumentN);

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

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

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

Calling Stored Procedures from JDBC

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

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

Calling Stored Procedures from SQLJ

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

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

// create the cursor


CustomerInfo customerinfo;

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

SQLIPSM

- All major database systems provide ways for users to write stored procedures in a simple, general purpose language closely aligned with SQL.

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

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

We can declare a function similarly as follows:

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

- Each parameter is a triple consisting of the mode the parameter name, and the SQL datatype of the parameter.
- The following SQL/PSM code computes the rating for a given customer and year.

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

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

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

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

ELSEIF statements;
ELSE statements; END IF
```

Loops are of the form

```
LOOP
        staternents:
END LOOP
```

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

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

CASE STUDY: THE INTERNET BOOK SHOP

They expect that the application logic will be implemented in Java, and so they consider JDBC and SQLJ as possible candidates for interfacing the database system with application code.

Recall that DBDudes settled on the following schema:

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

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

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

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

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

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

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

- Placing an order involves inserting one or more records into the Orders table.
- Since DBDudes has not yet chosen the Java-based technology to program the application logic, they assume for now that the individual books in the order are stored at the application layer in a Java array.
- For Example Order list Database:

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

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

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

Figure 6.11 Inserting a Completed Order into the Database

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

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

- The trigger processes each row in the order and calls the stored procedure 'UpdateShipDate'.

```
CREATE TRIGGER update_ShipDate
        AFTER INSERT ON Orders                    1* Event *j
    FOR EACH ROW
    BEGIN CALL UpdateShipDate(new); END           1* Action *j
```

Figure 6.12 Trigger to Update the Shipping Date of New Orders