Chapter 6: The basic SQL

6.1 SQL Data Definition and Data Types

SQL uses the terms table, row and column for the formal relational model terms *relation, tuple, and attribute*.

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, assertion and triggers.

6.1.1 Schema and Catalog Concepts in SQL

The concept of an SQL schema was incorporated starting with SQL2 in order to **group together tables** and **other constructs** that belong to the **same database application** (in some system. a *schema* is called a *database).*

An **SQL schema** is identified by a **schema name** and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema.

Schema elements include *tables, types, constraints, views, domains, and other constructs* (such as authorization grants) that describe the schema.

Can be created via the **CREATE SCHEMA,** element can be defined later.

```
CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';
```

NOTE: In general, not all user are authorized to create schemas, must be explicitly granted the relevant user account by the DBA or system administrator.

SQL use the concept of a **catalog -** a named collection of schemas.

A **catalog** always contains a *special schema* called **INFORMATION_SCHEMA** - which provides information on all the schemas in the catalog and all the element descriptors in these schema.

Integrity constraints such as **referential integrity** can be defined between relations only if they exist in schema with **same catalog.** And schema with the same catalog can share certain elements.

6.1.2 The CREATE TABLE Command in SQL

The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints.

The **attributes** are **specified first** and *each attribute is given a name, a data type* to specify its domain of values and *possibly attribute constraints*, such as **NOT NULL.**

The *key, entity integrity, and referential integrity constraints* can be specified within the **CREATE TABLE** statement after the attributes are declared, or they can be *added later* using the **ALTER TABLE** command.

The **SQL schema** in which the relations are declared is implicitly specified in the environment in which the **CREATE TABLE** statements are executed. Alternative, we can *explicitly attack the schema name to the relation name, separated by a period.*

```
CREATE TABLE COMPANY.EMPLOYEE
```

rather than

```
CREATE TABLE EMPLOYEE
```

```
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.

The relations declared through **CREATE TABLE** statements are called **base tables** (base relations) - means that **the table** and **its rows** are actually created and stored as a file by the DMBS.

Base relations are distinguished from **virtual relations**, create through the **CREATE VIEW** statement - which may or may not correspond to an actual physical file.

In **SQL,** the *attributes* in a **base tables** are considered to be *ordered in the sequence in which they are specified* in the **CREATE TABLE** statement. However, **rows (tuples)** are **not** considered to be *ordered in the sequence.*

6.1.3 Attribute Data Types and Domains in SQL

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

- **Numeric** data types include integer numbers of various sizes (**INTEGER or INT, and SMALLINT**) and floating-point (real) numbers of various precision (**FLOAT or REAL**, and **DOUBLE PRECISION**). **Formatted numbers** can be declared by using **DECIMAL(i, j)**—or **DEC(i, j)** or **NUMERIC(i, j)**—where ***i, the precision, is the total number of decimal digits*** and ***j, the scale, is the number of digits after the decimal point***. The **default for scale is zero**, and the **default for precision is implementation-defined**.
- **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**. For comparison purposes, strings are considered **ordered in alphabetic (or lexicographic) order**; if a string str1 appears before another string str2 in alphabetic order, then str1 is considered to be less than str2. There is **also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings**. For example, **'abc' || 'XYZ' results in a single string 'abcXYZ'**. Another variable-length string data type called **CHARACTER LARGE OBJECT**

or CLOB is also available to specify columns that have large text values, such as documents. The **CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G)**. For example, CLOB(20M) specifies a **maximum length of 20 megabytes**.

- **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'**. Another variable-length bit string 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 B**LOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits**.
- **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 t**hird possible value for a Boolean data type is UNKNOWN**.
- **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. The **< (less than) comparison can be used with dates or times—**an earlier date is considered to be smaller than a later date, and similarly with time. Literal values are represented by s**ingle quoted strings preceded by the keyword DATE or TIME**; for example, **DATE '2014-09-27' or TIME '09:12:47'**. In addition, a data type **TIME(i)**, where *i is called time fractional seconds precision, specifies i + 1 additional positions for TIME—one position for an additional period (.) separator character, and i positions for specifying decimal fractions of a second*. **A TIME WITH TIME ZONE** data type includes an additional s**ix positions for specifying the displacement from the standard universal time zone**, which is in the range **+13:00 to –12:59** in units

of **HOURS:MINUTES**. If **WITH TIME ZONE is not included**, the default is the local time zone for the SQL session.

- **Timestamp** data type (**TIMESTAMP)** include the **DATE** and **TIME fields,** plus a minimum of six positions for the decimal fraction of seconds and an optional **WITH TIME ZONE.** For example, **TIMESTAMP '2019-09-27 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**. Hence, they can generally be used in **string comparisons by being cast (or coerced or converted)** into the equivalent strings

It is possible to specify the data type of each attribute directly, as in Figure 6.1; alternatively, a **domain can be declared, and the domain name can be used with the attribute specification**. This makes it e**asier to change the data type** for a domain that is used by **numerous attributes** in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement.

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

In **SQL**, there is also a **CREATE TYPE** command, which can be used to create user defined types or UDTs. These can then be used either as **data types for attributes**, or as t**he basis for creating tables.**

6.2 Specifying Constraints in SQL

6.2.1 Specifying Attribute Constraints and Attribute Defaults

SQL allows **NULLs** as attribute values, a *constraint* **NOT NULL** may be specified if **NULL** is not permitted for a particular attribute. Always implicitly specified for the attributes that are part of the *primary key,* but i can be specified for any attributes whose values are *required not to be* **NULL.**

It is also possible to **define a default value** for an attribute by appending the clause **DEFAULT <value> to an attribute definition**. The **default value** is included in any new tuple if an explicit value is not provided for that attribute. 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.

Another type of constraint can restrict attribute or domain values using the CHECK clause following an attribute or domain definition. For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table (see Figure 6.1) to the following:

```
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);
```

6.2.2 Specifying Key and Referential Integrity Constraints

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

```
Dnumber INT PRIMARY KEY,
```

The **UNIQUE** clause specifies alternate (unique) keys, also known as candidate keys. 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.

Referential integrity is specified via the **FOREIGN KEY** clause, as shown in Figure 6.1. As we discussed in Section 5.2.4, a **referential integrity constraint** can be **violated when tuples are inserted or deleted**, or when a **foreign key or primary key attribute value is updated**. The d**efault action that SQL takes for an integrity violation** is to **reject the update operation** that will cause a violation, which is known as the **RESTRICT** option. However, the schema designer can specify an alternative action to be taken by attaching a referential triggered action clause to any foreign key constraint. The options include **SET NULL, CASCADE, and SET DEFAULT**. An option must be qualified with either **ON DELETE** or **ON UPDATE**. We illustrate this with the examples shown in Figure 6.2. Here, the database designer chooses **ON DELETE SET NULL and ON UPDATE CASCADE** for the foreign key Super_ssn of EMPLOYEE. This means that if the tuple for a supervising employee is **deleted**, the value of Super_ssn is **automatically set to NULL for all employee tuples that were referencing the deleted employee tuple**. On the other hand, if the **Ssn value for a supervising employee is updated** (say, because it was entered incorrectly), the **new value is cascaded to Super_ssn for all employee tuples referencing the updated employee tuple**.

6.2.3 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

6.2.4 Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other **table constraints** can be specified through additional **CHECK** clauses **at the end of a CREATE TABLE statement**. These can be called **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.

6.3 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**.

Before proceeding, we must **point out an important distinction between the practical SQL model and the formal relational model. SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values**. Hence, in general, an **SQL table is not a set of tuples, because a set does not allow two identical members**; rather, it is a **multiset** (sometimes called a **bag**) of tuples. Some **SQL relations are constrained to be sets** because **a key constraint has been declared or because the DISTINCT option has been used** with the **SELECT** statement.

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

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

```
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.

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';
```

The **SELECT** clause of **SQL** specifies the attributes whose values are to be retrieved, which are called **the projection attributes** in relational algebra

WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as **the selection condition** in relational algebra

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 conditio**n that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a **join condition**, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE.

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

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

```
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

6.2.2 Ambiguous Attribute Names, Aliasing, Renaming and Tuple Variables

In **SQL**, the **same name can be used** for two (or more) attributes as long as the attributes are in **different 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

```
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.

We can use this **alias-naming** or **renaming** mechanism in any **SQL** query to specify tuple variables for every table in the WHERE clause, whether or not the same relation needs to be referenced more than once.

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

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 claus**e, 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;
```

6.3.4 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.

194 Chapter 6 Basic SQL

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.

(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

An **SQL** table with a key is restricted to being a set, since the key value must be distinct in each tuple. If we **do want to eliminate duplicate tuples** from the result of an **SQL** query, we use the keyword **DISTINCT** in the **SELECT** clause, meaning that only distinct tuples should remain in the result. In general, a query with **SELECT DISTINCT** eliminates duplicates, whereas a query with **SELECT ALL** does not. Specifying **SELECT** with neither **ALL** nor **DISTINCT**—is equivalent to **SELECT ALL**.

SQL has **directly incorporated** some of the **set operations from mathematical set theory**, which are also part of relational algebra (see Chapter 8). There are **set union (UNION)**, **set difference (EXCEPT)**, and set **intersection (INTERSECT)** operations. The relations resulting from these set operations a**re sets of tuples**; that is, **duplicate tuples are eliminated from the result.** These set operations **apply only to type compatible relations**, so we must **make sure that the two relations on which we apply the operation have the same attributes** and that the **attributes appear in the same order in both relations**. The next example illustrates the use of UNION

(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. Notice that if several employees have the last name 'Smith', the project names involving any of them will be retrieved. Applying the **UNION** operation to the two SELECT queries gives the desired result.

SQL also has corresponding multiset operations, which are followed by the keyword A**LL (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.

6.3.5 Substring Pattern Matching and Arithmetic Operators

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 u**sing two reserved characters: %** replaces an **arbitrary number of zero or more characters**, and the **underscore (_) replaces a single characte**r. 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 placeholder 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 _ _ _ _ _ _ _';
```

If an **underscore** or **%** is needed as a **literal character** in the string, the character should be preceded by **an escape character**, which is **specified after the string** using the keyword **ESCAPE**. For example, 'AB_CD\%EF' ESCAPE '\' **represents the literal string 'AB_CD%EF' because \ is specified** as the **escape character.** Any **character not used in the string** can be chosen as the **escape character**. Also, we need a rule to specify **apostrophes or single quotation marks (' ')** if they are to be **included in a string** because they are used to begin and end strings. If an apostrophe (') is needed, it is represented as **two consecutive apostrophes ('')** so that it will not be interpreted as ending the string. *Notice that substring comparison implies that attribute values are not atomic (indivisible) values.*

The standard arithmetic operators for **addition (+), subtraction (–), multiplication (*), and division (/)** can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the **effect of giving all employees who work on the 'ProductX' project a 10% raise;** we can issue Query

13 to see what their salaries would become. This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

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 c**oncatenate operator || can be used in a query to append two string values**. For d**ate, 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)).

6.3.6 Ordering of Query Result

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
```

6.3.7 Discussion and Summary of Basic SQL Retrieval Queries

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

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

The SELECT clause lists the attributes to be retrieved, and the FROM clause specifies all relations (tables) needed in the simple query. The WHERE clause identifies the conditions for selecting the tuples from these relations, including

join conditions if needed. **ORDER BY** specifies an order for displaying the results of a query.

6.4 INSERT, DELETE and UPDATE Statements in SQL

INSERT is used to **add a single tuple (row) to a relation (table).** We must s**pecify 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.

```
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 DEFAULT values are the ones that can be left out**.

```
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. The attribute values forming each tuple are **enclosed in parentheses.**

```
INSERT INTO EMPLOYEE (Ssn, Fname, Lname, Dno, Salary)
VALUES
    ('111-22-3333', 'Alice', 'Johnson', 2, 55000),
    ('444-55-6666', 'Bob', 'Williams', 4, 60000),
    ('777-88-9999', 'Charlie', 'Brown', 1, 65000);
```

A **DBMS** that f**ully implements SQL** should **support and enforce all the integrity constraints** that can be specified in the **DDL**. For example, if we issue the command in U2 on the database shown in Figure 5.6, the **DBMS should reject the operation because no DEPARTMENT tuple exists in the database with Dnumber**

= 2. Similarly, **U2A would be rejected because no Ssn value is provided and it is the primary key**, which cannot be NULL

```
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**.

```
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;
```

Most **DBMSs** have **bulk loading tools that allow a user to load formatted data from a file into a table without having to write a large number of INSERT commands.**

Another variation for loading data is to create a new table TNEW that has the **same attributes** as an **existing table T**, and **load some of the data currently in T into TNEW**.

```
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.**

6.4.2 The DELETE Command

The **DELETE** command removes tuples from a relation. It includes a **WHERE** clause, similar to that used in an **SQL** query, to select the tuples to be deleted. **Tuples are explicitly deleted from only one table at a time**. However, **the deletion may propagate to tuples in other relations** if r**eferential 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 comman**d. **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 definitio**n.

```
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;
```

6.4.3 The UPDATE Command

The **UPDATE** command is used to **modify attribute values of one or more selected tuples**. As in the **DELETE** command, a **WHERE** clause in the **UPDATE** command **selects the tuples to be modified** from **a single relation**. However, **updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints** of the DDL (see Section 6.2.2). An additional **SET** clause in the **UPDATE** command **specifies the attributes to be modified and their new values**.

```
U5:    UPDATE    PROJECT
       SET       Plocation = 'Bellaire', Dnum = 5
       WHERE     Pnumber = 10;
```

Several tuples can be modified with a single UPDATE command.

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

It is also possible to specify **NULL** or **DEFAULT** as the **new attribute value**. *Notice that each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.*