chapter 4

Basic SQL

The SQL language may be considered one of the major reasons for the commercial success of relational databases. Because it became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems—for example, network or hierarchical systems—to relational systems. This is because even if the users became dissatisfied with the particular relational DBMS product they were using, converting to another relational DBMS product was not expected to be too expensive and time-consuming because both systems followed the same language standards. In practice, of course, there are many differences between various commercial relational DBMS packages. However, if the user is diligent in using only those features that are part of the standard, and if both relational systems faithfully support the standard, then conversion between the two systems should be much simplified. Another advantage of having such a standard is that users may write statements in a database application program that can access data stored in two or more relational DBMSs without having to change the database sublanguage (SQL) if both relational DBMSs support standard SQL.

This chapter presents the main features of the SQL standard for *commercial* relational DBMSs, whereas Chapter 3 presented the most important concepts underlying the *formal* relational data model. In Chapter 6 (Sections 6.1 through 6.5) we shall discuss the *relational algebra* operations, which are very important for understanding the types of requests that may be specified on a relational database. They are also important for query processing and optimization in a relational DBMS, as we shall see in Chapter 19. However, the relational algebra operations are considered to be too technical for most commercial DBMS users because a query in relational algebra is written as a sequence of operations that, when executed, produces the required result. Hence, the user must specify how—that is, *in what order*—to execute the query operations. On the other hand, the SQL language provides a

higher-level *declarative* language interface, so the user only specifies *what* the result is to be, leaving the actual optimization and decisions on how to execute the query to the DBMS. Although SQL includes some features from relational algebra, it is based to a greater extent on the *tuple relational calculus*, which we describe in Section 6.6. However, the SQL syntax is more user-friendly than either of the two formal languages.

The name **SQL** is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs. A joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) has led to a standard version of SQL (ANSI 1986), called SQL-86 or SQL1. A revised and much expanded standard called SQL-92 (also referred to as SQL2) was subsequently developed. The next standard that is well-recognized is SQL:1999, which started out as SQL3. Two later updates to the standard are SQL:2003 and SQL:2006, which added XML features (see Chapter 12) among other updates to the language. Another update in 2008 incorporated more object database features in SQL (see Chapter 11). We will try to cover the latest version of SQL as much as possible.

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL *and* a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java, COBOL, or C/C++.[1]

The later SQL standards (starting with **SQL:1999**) are divided into a **core** specification plus specialized **extensions**. The core is supposed to be implemented by all RDBMS vendors that are SQL compliant. The extensions can be implemented as optional modules to be purchased independently for specific database applications such as data mining, spatial data, temporal data, data warehousing, online analytical processing (OLAP), multimedia data, and so on.

Because SQL is very important (and quite large), we devote two chapters to its features. In this chapter, Section 4.1 describes the SQL DDL commands for creating schemas and tables, and gives an overview of the basic data types in SQL. Section 4.2 presents how basic constraints such as key and referential integrity are specified. Section 4.3 describes the basic SQL constructs for specifying retrieval queries, and Section 4.4 describes the SQL commands for insertion, deletion, and data updates.

In Chapter 5, we will describe more complex SQL retrieval queries, as well as the ALTER commands for changing the schema. We will also describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. We also introduce the concept of triggers, which is presented in

[1]Originally, SQL had statements for creating and dropping indexes on the files that represent relations, but these have been dropped from the SQL standard for some time.

more detail in Chapter 26 and we will describe the SQL facility for defining views on the database in Chapter 5. Views are also called *virtual* or *derived tables* because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables.

Section 4.5 lists some SQL features that are presented in other chapters of the book; these include transaction control in Chapter 21, security/authorization in Chapter 24, active databases (triggers) in Chapter 26, object-oriented features in Chapter 11, and online analytical processing (OLAP) features in Chapter 29. Section 4.6 summarizes the chapter. Chapters 13 and 14 discuss the various database programming techniques for programming with SQL.

4.1 SQL Data Definition and Data Types

SQL uses the terms **table**, **row**, and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively. We will use the corresponding terms interchangeably. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers). Before we describe the relevant CREATE statements, we discuss schema and catalog concepts in Section 4.1.1 to place our discussion in perspective. Section 4.1.2 describes how tables are created, and Section 4.1.3 describes the most important data types available for attribute specification. Because the SQL specification is very large, we give a description of the most important features. Further details can be found in the various SQL standards documents (see end-of-chapter bibliographic notes).

4.1.1 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. The concept of an SQL schema was incorporated starting with SQL2 in order to group together tables and other constructs that belong to the same database application. An **SQL schema** is identified by a **schema name**, and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema. Schema **elements** include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Alternatively, the schema can be assigned a name and authorization identifier, and the elements can be defined later. For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'. Note that each statement in SQL ends with a semicolon.

CREATE SCHEMA COMPANY **AUTHORIZATION** 'Jsmith';

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

In addition to the concept of a schema, SQL uses the concept of a **catalog**—a named collection of schemas in an SQL environment. An SQL **environment** is basically an installation of an SQL-compliant RDBMS on a computer system.[2] 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 schemas. Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog. Schemas within the same catalog can also share certain elements, such as domain definitions.

4.1.2 The CREATE TABLE Command in SQL

The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command (see Chapter 5). Figure 4.1 shows sample data definition statements in SQL for the COMPANY relational database schema shown in Figure 3.7.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. 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 4.1, 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 relation and its tuples are actually created and stored as a file by the DBMS. Base relations are distinguished from **virtual relations**, created through the CREATE VIEW statement (see Chapter 5), which may or may not correspond to an actual physical file. In SQL, the attributes in a base table 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 within a relation.

It is important to note that in Figure 4.1, there are some *foreign keys that may cause errors* because they are specified either via circular references or because they refer to a table that has not yet been created. For example, the foreign key Super_ssn in the EMPLOYEE table is a circular reference because it refers to the table itself. The foreign key Dno in the EMPLOYEE table refers to the DEPARTMENT table, which has

[2]SQL also includes the concept of a *cluster* of catalogs within an environment.

```
CREATE TABLE EMPLOYEE
        ( Fname             VARCHAR(15)         NOT NULL,
          Minit             CHAR,
          Lname             VARCHAR(15)         NOT NULL,
          Ssn               CHAR(9)             NOT NULL,
          Bdate             DATE,
          Address           VARCHAR(30),
          Sex               CHAR,
          Salary            DECIMAL(10,2),
          Super_ssn         CHAR(9),
          Dno               INT                 NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE DEPARTMENT
        ( Dname             VARCHAR(15)         NOT NULL,
          Dnumber           INT                 NOT NULL,
          Mgr_ssn           CHAR(9)             NOT NULL,
          Mgr_start_date    DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
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 4.1
SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 3.7.

not been created yet. 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 (see Chapter 5). We displayed all the foreign keys in Figure 4.1 to show the complete COMPANY schema in one place.

4.1.3 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).[3] For fixed-length strings, a shorter string is padded with blank characters to the right. For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith ' if needed. 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*.[4] There is also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings in SQL. 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

[3]This is not the case with SQL keywords, such as CREATE or CHAR. With keywords, SQL is *case insensitive*, meaning that SQL treats uppercase and lowercase letters as equivalent in keywords.

[4]For nonalphabetic characters, there is a defined order.

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.

- A **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN. We discuss the need for UNKNOWN and the three-valued logic in Chapter 5.
- 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 dates must be between 1 and 31; furthermore, a date should be a valid date 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 single-quoted strings preceded by the keyword DATE or TIME; for example, DATE '2008-09-27' 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 six 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.

Some additional data types are discussed below. The list of types discussed here is not exhaustive; different implementations have added more data types to SQL.

- A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single-quoted strings preceded by the keyword TIMESTAMP, with a blank space between data and time; for example, TIMESTAMP '2008-09-27 09:12:47.648302'.
- Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type. This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

[5]Bit strings whose length is a multiple of 4 can be specified in *hexadecimal* notation, where the literal string is preceded by X and each hexadecimal character represents 4 bits.

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

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

We can use SSN_TYPE in place of CHAR(9) in Figure 4.1 for the attributes Ssn and Super_ssn of EMPLOYEE, Mgr_ssn of DEPARTMENT, Essn of WORKS_ON, and Essn of DEPENDENT. A domain can also have an optional default specification via a DEFAULT clause, as we discuss later for attributes. Notice that domains may not be available in some implementations of SQL.

4.2 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. We discuss the specification of more general constraints, called assertions, in Chapter 5.

4.2.1 Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a *constraint* NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the *primary key* of each relation, but it can be specified for any other attributes whose values are required not to be NULL, as shown in Figure 4.1.

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 4.2 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.

Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition.[6] 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 4.1) to the following:

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

[6]The CHECK clause can also be used for other purposes, as we shall see.

```
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 4.2
Example illustrating how default attribute values and referential integrity triggered actions are specified in SQL.

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

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

We can then use the created domain D_NUM as the attribute type for all attributes that refer to department numbers in Figure 4.1, such as Dnumber of DEPARTMENT, Dnum of PROJECT, Dno of EMPLOYEE, and so on.

4.2.2 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. Some examples to illustrate the specification of keys and referential integrity are shown in Figure 4.1.[7] 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. For example, the primary key of DEPARTMENT can be specified as follows (instead of the way it is specified in Figure 4.1):

```
Dnumber INT PRIMARY KEY;
```

[7]Key and referential integrity constraints were not included in early versions of SQL. In some earlier implementations, keys were specified implicitly at the internal level via the CREATE INDEX command.

The **UNIQUE** clause specifies alternate (secondary) keys, as illustrated in the DEPARTMENT and PROJECT table declarations in Figure 4.1. The **UNIQUE** clause can also be specified directly for a secondary key if the secondary key 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 4.1. As we discussed in Section 3.2.4, a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified. The default action that SQL takes for an integrity violation is to **reject** the update operation that will cause a violation, which is known as the RESTRICT option. However, the schema designer can specify an alternative action to be taken by attaching a **referential triggered action** clause to any foreign key constraint. The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE. We illustrate this with the examples shown in Figure 4.2. 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.[8]

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. It is the responsibility of the database designer to choose the appropriate action and to specify it in the database schema. As a general rule, the CASCADE option is suitable for "relationship" relations (see Section 9.1), such as WORKS_ON; for relations that represent multivalued attributes, such as DEPT_LOCATIONS; and for relations that represent weak entity types, such as DEPENDENT.

4.2.3 Giving Names to Constraints

Figure 4.2 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 con-

[8]Notice that the foreign key Super_ssn in the EMPLOYEE table is a circular reference and hence may have to be added later as a named constraint using the ALTER TABLE statement as we discussed at the end of Section 4.1.2.

straint in case the constraint must be dropped later and replaced with another constraint, as we discuss in Chapter 5. Giving names to constraints is optional.

4.2.4 Specifying Constraints on Tuples Using CHECK

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

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

The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL. We discuss this in Chapter 5 because it requires the full power of queries, which are discussed in Sections 4.3 and 5.1.

4.3 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, which we discuss in Chapter 6. There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually. We will use sample queries specified on the schema of Figure 3.5 and will refer to the sample database state shown in Figure 3.6 to show the results of some of the sample queries. In this section, we present the features of SQL for *simple retrieval queries*. Features of SQL for specifying more complex retrieval queries are presented in Section 5.1.

Before proceeding, we must point out an *important distinction* between SQL and the formal relational model discussed in Chapter 3: 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 (described later in this section). We should be aware of this distinction as we discuss the examples.

4.3.1 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:[9]

```
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.

We illustrate the basic SELECT statement in SQL with some sample queries. The queries are labeled here with the same query numbers used in Chapter 6 for easy cross-reference.

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.

The SELECT clause of SQL specifies the attributes whose values are to be retrieved, which are called the **projection attributes**, and the WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the **selection condition**. Figure 4.3(a) shows the result of query Q0 on the database of Figure 3.6.

We can think of an implicit **tuple variable** or *iterator* in the SQL query ranging or *looping* over each individual tuple in the EMPLOYEE table and evaluating the condition in the WHERE clause. Only those tuples that satisfy the condition—that is,

[9]The SELECT and FROM clauses are required in all SQL queries. The WHERE is optional (see Section 4.3.3).

Figure 4.3
Results of SQL queries when applied to the COMPANY database state shown in Figure 3.6. (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

those tuples for which the condition evaluates to TRUE after substituting their corresponding attribute values—are selected.

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 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. The result of query Q1 is shown in Figure 4.3(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';
```

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, and one employee that satisfies the join conditions. 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 4.3(c).

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

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in *different relations.* If this is the case, and a multitable query refers to two or more attributes with the same name, we *must* **qualify** the attribute name with the relation name to prevent ambiguity. This is done by *prefixing* the relation name to the attribute name and separating the two by a period. To illustrate this, suppose that in Figures 3.5 and 3.6 the Dno and Lname attributes of the EMPLOYEE relation were

called Dnumber and Name, and the Dname attribute of DEPARTMENT was also called Name; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A. We must prefix the attributes Name and Dnumber in Q1A to specify which ones we are referring to, because the same attribute names are used in both relations:

```
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 is shown in this manner as is Q1′ below. We can also create 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. We can now join the two copies. Of course, in reality there is *only one* EMPLOYEE relation, and the join condition is meant to join the relation with itself by matching the tuples that satisfy the join condition E.Super_ssn = S.Ssn. Notice that this is an example of a one-level recursive query, as we will discuss in Section 6.4.2. In earlier versions of SQL, it was not possible to specify a general recursive query, with an unknown number of levels, in a

single SQL statement. A construct for specifying recursive queries has been incorporated into SQL:1999 (see Chapter 5).

The result of query Q8 is shown in Figure 4.3(d). Whenever one or more aliases are given to a relation, we can use these names to represent different references to that same relation. This permits multiple references to the same relation within a query.

We can use this alias-naming 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. In fact, this practice is recommended since it results in queries that are easier to comprehend. For example, we could specify query Q1 as in Q1B:

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

4.3.3 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. For example, Query 9 selects all EMPLOYEE Ssns (Figure 4.3(e)), and Query 10 selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname, regardless of whether the employee works for the department or not (Figure 4.3(f)).

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. Notice that Q10 is similar to a CROSS PRODUCT operation followed by a PROJECT operation in relational algebra (see Chapter 6). If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q10, we get the actual CROSS PRODUCT (except for duplicate elimination, if any).

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.* For example, query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 4.3(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;
```

4.3.4 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 5.1.7) is applied to tuples, in most cases we do not want to eliminate duplicates.

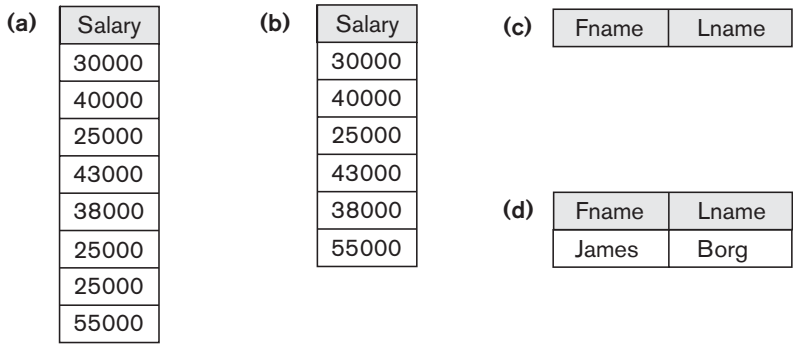
An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple.[10] If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not. Specifying SELECT with neither ALL nor DISTINCT—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 4.4(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 4.4(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;
```

[10] In general, an SQL table is not required to have a key, although in most cases there will be one.



(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 4.4
Results of additional SQL queries when applied to the COMPANY database state shown in Figure 3.6. (a) Q11. (b) Q11A. (c) Q16. (d) Q18.

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra (see Chapter 6). There are set union (UNION), set difference (**EXCEPT**),[11] and set intersection (**INTERSECT**) operations. The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result.* These set operations apply only to *union-compatible relations*, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations. The next example illustrates the use of UNION.

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

```
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 **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 4.5. Basically, each tuple—whether it is a duplicate or not—is considered as a different tuple when applying these operations.

[11]In some systems, the keyword MINUS is used for the set difference operation instead of EXCEPT.

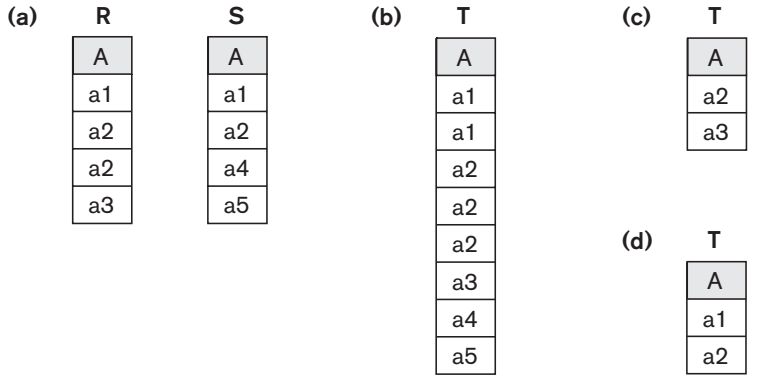


Figure 4.5
The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

4.3.5 Substring Pattern Matching and Arithmetic Operators

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

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

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

To retrieve all employees who were born during the 1950s, we can use Query Q12A. Here, '5' 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 '_ _ 5 _ _ _ _ _ _ _';
```

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, as we had assumed in the formal relational model (see Section 3.1).

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

4.3.6 Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the **ORDER BY** clause. 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 D, EMPLOYEE E, WORKS_ON W,
                 PROJECT 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**

4.3.7 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. Two additional clauses GROUP BY and HAVING will be described in Section 5.1.8.

In Chapter 5, we will present more complex features of SQL retrieval queries. These include the following: nested queries that allow one query to be included as part of another query; aggregate functions that are used to provide summaries of the information in the tables; two additional clauses (GROUP BY and HAVING) that can be used to provide additional power to aggregate functions; and various types of joins that can combine records from various tables in different ways.

4.4 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.

4.4.1 The INSERT Command

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

in Figure 3.5 and specified in the CREATE TABLE EMPLOYEE ... command in Figure 4.1, 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 DEFAULT values 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. The attribute values forming *each tuple* are enclosed in parentheses.

A DBMS that fully 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 3.6, 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.

```
U3:   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. We can now query WORKS_ON_INFO as we would any other relation; when we do not need it any more, we can remove it by using the DROP TABLE command (see Chapter 5). Notice that the WORKS_ON_INFO table may not be up-to-date; that is, if we update any of the PROJECT, WORKS_ON, or EMPLOYEE relations after issuing U3B, the information in WORKS_ON_INFO *may become outdated*. We have to create a view (see Chapter 5) to keep such a table up-to-date.

4.4.2 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 *referential triggered actions* are specified in the referential integrity constraints of the DDL (see Section 4.2.2).[12] 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 (see Chapter 5). The DELETE commands in U4A to U4D, if applied independently to the database in Figure 3.6, 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;
```

4.4.3 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

[12]Other actions can be automatically applied through triggers (see Section 26.1) and other mechanisms.

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 4.2.2). 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 percent 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;
```

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.

4.5 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:

- In Chapter 5, which is a continuation of this chapter, we will present the following SQL features: various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification.
- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Persistent Stored Modules). We discuss these techniques in Chapter 13. We also discuss how to access SQL databases through the Java programming language using JDBC and SQLJ.
- 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. We called these commands a *storage definition language* (*SDL*) in Chapter 2. Earlier versions of SQL had commands for **creating indexes**, but these were removed from the

we will present the following features of SQL: complex retrieval queries; views; triggers and assertions; and schema modification commands.

Review Questions

4.1. How do the relations (tables) in SQL differ from the relations defined formally in Chapter 3? Discuss the other differences in terminology. Why does SQL allow duplicate tuples in a table or in a query result?

4.2. List the data types that are allowed for SQL attributes.

4.3. How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 3? What about referential triggered actions?

4.4. Describe the four clauses in the syntax of a simple SQL retrieval query. Show what type of constructs can be specified in each of the clauses. Which are required and which are optional?

Exercises

4.5. Consider the database shown in Figure 1.2, whose schema is shown in Figure 2.1. What are the referential integrity constraints that should hold on the schema? Write appropriate SQL DDL statements to define the database.

4.6. Repeat Exercise 4.5, but use the AIRLINE database schema of Figure 3.8.

4.7. Consider the LIBRARY relational database schema shown in Figure 4.6. Choose the appropriate action (reject, cascade, set to NULL, set to default) for each referential integrity constraint, both for the *deletion* of a referenced tuple and for the *update* of a primary key attribute value in a referenced tuple. Justify your choices.

4.8. Write appropriate SQL DDL statements for declaring the LIBRARY relational database schema of Figure 4.6. Specify the keys and referential triggered actions.

4.9. How can the key and foreign key constraints be enforced by the DBMS? Is the enforcement technique you suggest difficult to implement? Can the constraint checks be executed efficiently when updates are applied to the database?

4.10. Specify the following queries in SQL on the COMPANY relational database schema shown in Figure 3.5. Show the result of each query if it is applied to the COMPANY database in Figure 3.6.

a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.

b. List the names of all employees who have a dependent with the same first name as themselves.

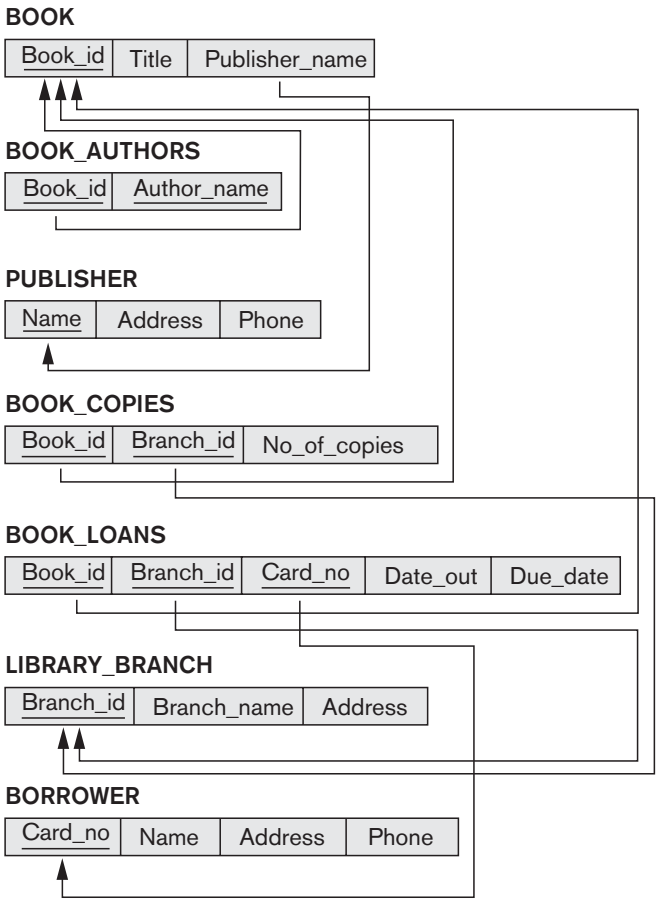


Figure 4.6
A relational database schema for a LIBRARY database.

c. Find the names of all employees who are directly supervised by 'Franklin Wong'.

4.11. Specify the updates of Exercise 3.11 using the SQL update commands.

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

a. Retrieve the names of all senior students majoring in 'CS' (computer science).

b. Retrieve the names of all courses taught by Professor King in 2007 and 2008.

c. For each section taught by Professor King, retrieve the course number, semester, year, and number of students who took the section.

d. Retrieve the name and transcript of each senior student (Class = 4) majoring in CS. A transcript includes course name, course number, credit hours, semester, year, and grade for each course completed by the student.

4.13. Write SQL update statements to do the following on the database schema shown in Figure 1.2.

a. Insert a new student, <'Johnson', 25, 1, 'Math'>, in the database.

b. Change the class of student 'Smith' to 2.

c. Insert a new course, <'Knowledge Engineering', 'CS4390', 3, 'CS'>.

d. Delete the record for the student whose name is 'Smith' and whose student number is 17.

4.14. Design a relational database schema for a database application of your choice.

a. Declare your relations, using the SQL DDL.

b. Specify a number of queries in SQL that are needed by your database application.

c. Based on your expected use of the database, choose some attributes that should have indexes specified on them.

d. Implement your database, if you have a DBMS that supports SQL.

4.15. Consider the EMPLOYEE table's constraint EMPSUPERFK as specified in Figure 4.2 is changed to read as follows:

```
CONSTRAINT EMPSUPERFK
  FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                ON DELETE CASCADE ON UPDATE CASCADE,
```

Answer the following questions:

a. What happens when the following command is run on the database state shown in Figure 3.6?

```
DELETE EMPLOYEE WHERE Lname = 'Borg'
```

b. Is it better to CASCADE or SET NULL in case of EMPSUPERFK constraint ON DELETE?

4.16. Write SQL statements to create a table EMPLOYEE_BACKUP to back up the EMPLOYEE table shown in Figure 3.6.

Selected Bibliography

The SQL language, originally named SEQUEL, was based on the language SQUARE (Specifying Queries as Relational Expressions), described by Boyce et al. (1975). The syntax of SQUARE was modified into SEQUEL (Chamberlin and Boyce, 1974) and then into SEQUEL 2 (Chamberlin et al. 1976), on which SQL is based. The original implementation of SEQUEL was done at IBM Research, San Jose, California. We will give additional references to various aspects of SQL at the end of Chapter 5.

chapter 5

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

This chapter describes more advanced features of the SQL language standard for relational databases. We start in Section 5.1 by presenting more complex features of SQL retrieval queries, such as nested queries, joined tables, outer joins, aggregate functions, and grouping. In Section 5.2, we describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. We also introduce the concept of triggers and the CREATE TRIGGER statement, which will be presented in more detail in Section 26.1 when we present the principles of active databases. Then, in Section 5.3, we describe the SQL facility for defining views on the database. Views are also called *virtual* or *derived tables* because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables. Section 5.4 introduces the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints. Section 5.5 is the chapter summary.

This chapter is a continuation of Chapter 4. The instructor may skip parts of this chapter if a less detailed introduction to SQL is intended.

5.1 More Complex SQL Retrieval Queries

In Section 4.3, we described some basic types of retrieval queries in SQL. Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex retrievals from the database. We discuss several of these features in this section.

5.1.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. Recall from Section 3.1.2 that NULL is used to represent a missing value, but that it usually has one of three different interpretations—value *unknown* (exists but is not known), value *not available* (exists but is purposely withheld), or value *not applicable* (the attribute is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL.

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

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

In general, each individual NULL value is considered to be different from every other NULL value in the various database records. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used. Table 5.1 shows the resulting values.

Table 5.1 Logical Connectives in Three-Valued Logic

(a)

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

(b)

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

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

In Tables 5.1(a) and 5.1(b), the rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query. Each expression result would have a value of TRUE, FALSE, or UNKNOWN. The result of combining the two values using the AND logical connective is shown by the entries in Table 5.1(a). Table 5.1(b) shows the result of using the OR logical connective. For example, the result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN. Table 5.1(c) shows the result of the NOT logical operation. Notice that in standard Boolean logic, only TRUE or FALSE values are permitted; there is no UNKNOWN value.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected. However, there are exceptions to that rule for certain operations, such as outer joins, as we shall see in Section 5.1.6.

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **IS** or **IS NOT**. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN; see Section 5.1.6). Query 18 illustrates this.

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

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

5.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

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

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

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

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

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

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

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

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

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

Notice that this query can also be specified using the MAX aggregate function (see Section 5.1.7).

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

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

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

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

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

5.1.3 Correlated Nested Queries

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

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

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

5.1.4 The EXISTS and UNIQUE Functions in SQL

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

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

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

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

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

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

empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

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

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

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

The query Q3: *Retrieve the name of each employee who works on* all *the projects controlled by department number 5* can be written using EXISTS and NOT EXISTS in SQL systems. We show two ways of specifying this query Q3 in SQL as Q3A and Q3B. This is an example of certain types of queries that require *universal quantification*, as we will discuss in Section 6.6.7. One way to write this query is to use the construct (*S2* EXCEPT *S1*) as explained next, and checking whether the result is empty.[1] This option is shown as Q3A.

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

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

The second option is shown as Q3B. Notice that we need two-level nesting in Q3B and that this formulation is quite a bit more complex than Q3A, which uses NOT EXISTS and EXCEPT.

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

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

In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose Pno is of a project controlled by department 5, *if* there is not a WORKS_ON (C) tuple with the same Pno and the same Ssn as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on. It corresponds to the way we will write this query in tuple relation calculus (see Section 6.6.7).

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

5.1.5 Explicit Sets and Renaming of Attributes 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, and it can be used in both the SELECT and FROM clauses. 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 in 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;
```

5.1.6 Joined Tables in SQL and Outer Joins

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

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

The FROM clause in Q1A contains a single *joined table.* The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a NATURAL JOIN on two relations *R* and *S,* no join condition is specified; an implicit *EQUIJOIN condition* for *each pair of attributes with the same name* from *R* and *S* is created. Each such pair of attributes is included *only once* in the resulting relation (see Section 6.3.2 and 6.4.4 for more details on the various types of join operations in relational algebra).

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

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

The default type of join in a joined table is called an **inner join**, where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in query Q8A, only employees who *have a supervisor* are included in the result; an EMPLOYEE tuple whose value for Super_ssn is NULL is excluded. If the user requires that all employees be included, an OUTER JOIN must be used explicitly (see Section 6.4.4 for the definition of OUTER JOIN). In SQL, 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);
```

There are a variety of outer join operations, which we shall discuss in more detail in Section 6.4.4. In SQL, the options available for specifying joined tables include INNER JOIN (only pairs of tuples that match the join condition are retrieved, same as JOIN), LEFT OUTER JOIN (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table), RIGHT OUTER JOIN (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table), and FULL OUTER JOIN. In the latter three options, the keyword OUTER may be omitted. If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN). The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation (see Section 6.2.2), although this should be used only with the utmost care because it generates all possible tuple combinations.

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**. For example, Q2A is a different way of specifying query Q2 from Section 4.3.1 using the concept of a joined table:

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

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

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

5.1.7 Aggregate Functions in SQL

In Section 6.4.2, we will introduce the concept of an aggregate function as a relational algebra operation. **Aggregate functions** are used to summarize information from multiple tuples into a single-tuple summary. **Grouping** is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database applications, and we will introduce their use in SQL through examples. A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.[2] The COUNT function returns the number of tuples or values as specified in a

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

query. The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.[3] We illustrate the use of these functions with sample 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;
```

If we want to get the preceding 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;
```

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY

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

will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute).

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

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

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

5.1.8 Grouping: The GROUP BY and HAVING Clauses

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

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

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

In Q24, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the grouping attribute Dno. Hence, each group contains the employees who work in the same department. The COUNT and AVG functions are applied to each such group of tuples. Notice that the SELECT clause includes only the grouping attribute and the aggregate functions to be applied on each group of tuples. Figure 5.1(a) illustrates how grouping works on Q24; it also shows the result of Q24.

Figure 5.1
Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

(a)

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

Grouping EMPLOYEE tuples by the value of Dno

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

Result of Q24

(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)

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

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

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

Notice that while selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups.* Figure 5.1(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;
```

Here we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE clause—namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the aggregate function in the SELECT clause and another to the function in the HAVING clause). For example, suppose that we want

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

```
SELECT    Dname, COUNT (*)
FROM      DEPARTMENT, EMPLOYEE
WHERE     Dnumber=Dno AND Salary>40000
GROUP BY  Dname
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. Hence, 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    Dnumber, COUNT (*)
      FROM      DEPARTMENT, EMPLOYEE
      WHERE     Dnumber=Dno AND Salary>40000 AND
                ( SELECT        Dno
                  FROM          EMPLOYEE
                  GROUP BY Dno
                  HAVING        COUNT (*) > 5)
```

5.1.9 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two—SELECT and FROM—are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

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

The SELECT clause lists the attributes or functions to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. GROUP BY

specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

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

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

The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way. Ideally, this should not be the case: The DBMS should process the same query in the same way regardless of how the query is specified. But this is quite difficult in practice, since each DBMS has different methods for processing queries specified in different ways. Thus, an additional burden on the user is to determine which of the alternative specifications is the most efficient to execute. Ideally, the user should worry only about specifying the query correctly, whereas the DBMS would determine how to execute the query efficiently. In practice, however, it helps if the user is aware of which types of constructs in a query are more expensive to process than others (see Chapter 20).

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

Review Questions

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

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

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

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

a. Nested queries.
b. Joined tables and outer joins.
c. Aggregate functions and grouping.
d. Triggers.
e. Assertions and how they differ from triggers.
f. Views and their updatability.
g. Schema change commands.

Exercises

5.5. Specify the following queries on the database in Figure 3.5 in SQL. Show the query results if each query is applied to the database in Figure 3.6.

a. For each department whose average employee salary is more than $30,000, retrieve the department name and the number of employees working for that department.
b. Suppose that we want the number of *male* employees in each department making more than $30,000, rather than all employees (as in Exercise 5.4a). Can we specify this query in SQL? Why or why not?

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

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

5.7. In SQL, specify the following queries on the database in Figure 3.5 using the concept of nested queries and concepts described in this chapter.

a. Retrieve the names of all employees who work in the department that has the employee with the highest salary among all employees.
b. Retrieve the names of all employees whose supervisor's supervisor has '888665555' for Ssn.

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

5.8. Specify the following views in SQL on the COMPANY database schema shown in Figure 3.5.

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

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

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

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

5.9. Consider the following view, DEPT_SUMMARY, defined on the COMPANY database in Figure 3.6:

```
CREATE VIEW   DEPT_SUMMARY (D, C, Total_s, Average_s)
AS SELECT     Dno, COUNT (*), SUM (Salary), AVG (Salary)
FROM          EMPLOYEE
GROUP BY      Dno;
```

State which of the following queries and updates would be allowed on the view. If a query or update would be allowed, show what the corresponding query or update on the base relations would look like, and give its result when applied to the database in Figure 3.6.

a.
```
SELECT   *
FROM     DEPT_SUMMARY;
```

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

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

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

e.
```
DELETE   FROM DEPT_SUMMARY
WHERE    C > 4;
```

matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

This informal guideline obviously needs to be stated more formally. In Section 16.2 we discuss a formal condition called the nonadditive (or lossless) join property that guarantees that certain joins do not produce spurious tuples.

15.1.5 Summary and Discussion of Design Guidelines

In Sections 15.1.1 through 15.1.4, we informally discussed situations that lead to problematic relation schemas and we proposed informal guidelines for a good relational design. The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values
- Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

In the rest of this chapter we present formal concepts and theory that may be used to define the *goodness* and *badness* of *individual* relation schemas more precisely. First we discuss functional dependency as a tool for analysis. Then we specify the three normal forms and Boyce-Codd normal form (BCNF) for relation schemas. The strategy for achieving a good design is to decompose a badly designed relation appropriately. We also briefly introduce additional normal forms that deal with additional dependencies. In Chapter 16, we discuss the properties of decomposition in detail, and provide algorithms that design relations bottom-up by using the functional dependencies as a starting point.

15.2 Functional Dependencies

So far we have dealt with the informal measures of database design. We now introduce a formal tool for analysis of relational schemas that enables us to detect and describe some of the above-mentioned problems in precise terms. The single most important concept in relational schema design theory is that of a functional dependency. In this section we formally define the concept, and in Section 15.3 we see how it can be used to define normal forms for relation schemas.

15.2.1 Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes A_1, A_2, ..., A_n; let us think of the whole database as being described by a single **universal**

relation schema $R = \{A_1, A_2, \ldots, A_n\}$.[7] We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.[8]

> **Definition.** A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint* on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are *determined by,* the values of the X component; alternatively, the values of the X component of a tuple uniquely (or **functionally**) *determine* the values of the Y component. We also say that there is a functional dependency from X to Y, or that Y is **functionally dependent** on X. The abbreviation for functional dependency is **FD** or **f.d.** The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side**.

Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of $r(R)$ agree on their X-value, they must necessarily agree on their Y-value. Note the following:

- If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation instance $r(R)$—that is, X is a **candidate key** of R—this implies that $X \rightarrow Y$ for any subset of attributes Y of R (because the key constraint implies that no two tuples in any legal state $r(R)$ will have the same value of X). If X is a candidate key of R, then $X \rightarrow R$.
- If $X \rightarrow Y$ in R, this does not say whether or not $Y \rightarrow X$ in R.

A functional dependency is a property of the **semantics** or **meaning of the attributes**. The database designers will use their understanding of the semantics of the attributes of R—that is, how they relate to one another—to specify the functional dependencies that should hold on *all* relation states (extensions) r of R. Whenever the semantics of two sets of attributes in R indicate that a functional dependency should hold, we specify the dependency as a constraint. Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of R. Hence, the main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold *at all times.* Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning. For example, {State, Driver_license_number} $\rightarrow$ Ssn should hold for any adult in the United States and hence should hold whenever these attributes appear in a relation. It is also possible that certain functional dependencies may cease to

[7]This concept of a universal relation is important when we discuss the algorithms for relational database design in Chapter 16.

[8]This assumption implies that every attribute in the database should have a distinct name. In Chapter 3 we prefixed attribute names by relation names to achieve uniqueness whenever attributes in distinct relations had the same name.

exist in the real world if the relationship changes. For example, the FD Zip_code → Area_code used to exist as a relationship between postal codes and telephone number codes in the United States, but with the proliferation of telephone area codes it is no longer true.

Consider the relation schema EMP_PROJ in Figure 15.3(b); from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

a. Ssn → Ename

b. Pnumber →{Pname, Plocation}

c. {Ssn, Pnumber} → Hours

These functional dependencies specify that (a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename), (b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and (c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours). Alternatively, we say that Ename is functionally determined by (or functionally dependent on) Ssn, or *given a value of Ssn, we know the value of Ename,* and so on.

A functional dependency is a *property of the relation schema R*, not of a particular legal relation state *r* of *R*. Therefore, an FD *cannot* be inferred automatically from a given relation extension *r* but must be defined explicitly by someone who knows the semantics of the attributes of *R*. For example, Figure 15.7 shows a particular state of the TEACH relation schema. Although at first glance we may think that Text → Course, we cannot confirm this unless we know that it is true *for all possible legal states* of TEACH. It is, however, sufficient to demonstrate *a single counterexample* to disprove a functional dependency. For example, because 'Smith' teaches both 'Data Structures' and 'Data Management,' we can conclude that Teacher *does not* functionally determine Course.

Given a populated relation, one cannot determine which FDs hold and which do not unless the meaning of and the relationships among the attributes are known. All one can say is that a certain FD *may* exist if it holds in that particular extension. One cannot guarantee its existence until the meaning of the corresponding attributes is clearly understood. One can, however, emphatically state that a certain FD *does not*

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Figure 15.7
A relation state of TEACH with a *possible* functional dependency TEXT → COURSE. However, TEACHER → COURSE is ruled out.

hold if there are tuples that show the violation of such an FD. See the illustrative example relation in Figure 15.8. Here, the following FDs *may hold* because the four tuples in the current extension have no violation of these constraints: $B \rightarrow C$; $C \rightarrow B$; $\{A, B\} \rightarrow C$; $\{A, B\} \rightarrow D$; and $\{C, D\} \rightarrow B$. However, the following *do not* hold because we already have violations of them in the given extension: $A \rightarrow B$ (tuples 1 and 2 violate this constraint); $B \rightarrow A$ (tuples 2 and 3 violate this constraint); $D \rightarrow C$ (tuples 3 and 4 violate it).

Figure 15.3 introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, while the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

We denote by F the set of functional dependencies that are specified on relation schema R. Typically, the schema designer specifies the functional dependencies that are *semantically obvious*; usually, however, numerous other functional dependencies hold in *all* legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in F. Those other dependencies can be *inferred* or *deduced* from the FDs in F. We defer the details of inference rules and properties of functional dependencies to Chapter 16.

15.3 Normal Forms Based on Primary Keys

Having introduced functional dependencies, we are now ready to use them to specify some aspects of the semantics of relation schemas. We assume that a set of functional dependencies is given for each relation, and that each relation has a designated primary key; this information combined with the tests (conditions) for normal forms drives the *normalization process* for relational schema design. Most practical relational design projects take one of the following two approaches:

- Perform a conceptual schema design using a conceptual model such as ER or EER and map the conceptual design into a set of relations
- Design the relations based on external knowledge derived from an existing implementation of files or forms or reports

Following either of these approaches, it is then useful to evaluate the relations for goodness and decompose them further as needed to achieve higher normal forms, using the normalization theory presented in this chapter and the next. We focus in

Figure 15.8
A relation R(A, B, C, D) with its extension.

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

this section on the first three normal forms for relation schemas and the intuition behind them, and discuss how they were developed historically. More general definitions of these normal forms, which take into account all candidate keys of a relation rather than just the primary key, are deferred to Section 15.4.

We start by informally discussing normal forms and the motivation behind their development, as well as reviewing some definitions from Chapter 3 that are needed here. Then we discuss the first normal form (1NF) in Section 15.3.4, and present the definitions of second normal form (2NF) and third normal form (3NF), which are based on primary keys, in Sections 15.3.5 and 15.3.6, respectively.

15.3.1 Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**. The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as *relational design by analysis.* Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively; these are briefly discussed in Sections 15.6 and 15.7.

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies discussed in Section 15.1.2. It can be considered as a "filtering" or "purification" process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database designers with the following:

- A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes
- A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree

Definition. The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

Normal forms, when considered *in isolation* from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each

relation schema in the database is, say, in BCNF or 3NF. Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:

- The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem discussed in Section 15.1.4 does not occur with respect to the relation schemas created after decomposition.
- The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

The nonadditive join property is extremely critical and **must be achieved at any cost**, whereas the dependency preservation property, although desirable, is sometimes sacrificed, as we discuss in Section 16.1.2. We defer the presentation of the formal concepts and techniques that guarantee the above two properties to Chapter 16.

15.3.2 Practical Use of Normal Forms

Most practical design projects acquire existing designs of databases from previous designs, designs in legacy models, or from existing files. Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. Although several higher normal forms have been defined, such as the 4NF and 5NF that we discuss in Sections 15.6 and 15.7, the practical utility of these normal forms becomes questionable when the constraints on which they are based are rare, and hard to understand or to detect by the database designers and users who must discover these constraints. Thus, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.

Another point worth noting is that the database designers *need not* normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reasons, such as those discussed at the end of Section 15.1.2. Doing so incurs the corresponding penalties of dealing with the anomalies.

> **Definition. Denormalization** is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

15.3.3 Definitions of Keys and Attributes Participating in Keys

Before proceeding further, let's look again at the definitions of keys of a relation schema from Chapter 3.

> **Definition.** A **superkey** of a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more.

The difference between a key and a superkey is that a key has to be *minimal;* that is, if we have a key $K = \{A_1, A_2, ..., A_k\}$ of R, then $K - \{A_i\}$ is not a key of R for any A_i, $1 \leq i \leq k$. In Figure 15.1, {Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename}, {Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys.

If a relation schema has more than one key, each is called a **candidate key**. One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called secondary keys. In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In Figure 15.1, {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary key.

> **Definition.** An attribute of relation schema R is called a **prime attribute** of R if it is a member of *some candidate key* of R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

In Figure 15.1, both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

We now present the first three normal forms: 1NF, 2NF, and 3NF. These were proposed by Codd (1972a) as a sequence to achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed. As we shall see, 2NF and 3NF attack different problems. However, for historical reasons, it is customary to follow them in that sequence; hence, by definition a 3NF relation *already satisfies* 2NF.

15.3.4 First Normal Form

First normal form (**1NF**) is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple.* In other words, 1NF disallows *relations within relations* or *relations as attribute values within tuples*. The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

Consider the DEPARTMENT relation schema shown in Figure 15.1, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in Figure 15.9(a). We assume that each department can have *a number of* locations. The DEPARTMENT schema and a sample relation state are shown in Figure 15.9. As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure 15.9(b). There are two ways we can look at the Dlocations attribute:

- The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber.

(a)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations

(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 15.9
Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

- The domain of Dlocations contains sets of values and hence is nonatomic. In this case, Dnumber → Dlocations because each set is considered a single member of the attribute domain.[9]

In either case, the DEPARTMENT relation in Figure 15.9 is not in 1NF; in fact, it does not even qualify as a relation according to our definition of relation in Section 3.1. There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute Dlocations that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this relation is the combination {Dnumber, Dlocation}, as shown in Figure 15.2. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.

[9]In this case we can consider the domain of Dlocations to be the **power set** of the set of single locations; that is, the domain is made up of all possible subsets of the set of single locations.

2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 15.9(c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing *redundancy* in the relation.
3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing *NULL values* if most departments have fewer than three locations. It further introduces spurious semantics about the ordering among the location values that is not originally intended. Querying on this attribute becomes more difficult; for example, consider how you would write the query: *List the departments that have 'Bellaire' as one of their locations* in this design.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it.* Figure 15.10 shows how the EMP_PROJ relation could appear if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS(Pnumber, Hours) *within each tuple* represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

EMP_PROJ(Ssn, Ename, {PROJS(Pnumber, Hours)})

The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses ( ). Interestingly, recent trends for supporting complex objects (see Chapter 11) and XML data (see Chapter 12) attempt to allow and formalize nested relations within relational database systems, which were disallowed early on by 1NF.

Notice that Ssn is the primary key of the EMP_PROJ relation in Figures 15.10(a) and (b), while Pnumber is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of Pnumber. To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2, as shown in Figure 15.10(c).

This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations. The

(a)

EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

(b)

EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

(c)

EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

Figure 15.10
Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a *nested relation* attribute PROJS. (b) Sample extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

existence of more than one multivalued attribute in one relation must be handled carefully. As an example, consider the following non-1NF relation:

PERSON (Ss#, {Car_lic#}, {Phone#})

This relation represents the fact that a person has multiple cars and multiple phones. If strategy 2 above is followed, it results in an all-key relation:

PERSON_IN_1NF (Ss#, Car_lic#, Phone#)

To avoid introducing any extraneous relationship between Car_lic# and Phone#, all possible combinations of values are represented for every Ss#, giving rise to redundancy. This leads to the problems handled by multivalued dependencies and 4NF, which we will discuss in Section 15.6. The right way to deal with the two multivalued attributes in PERSON shown previously is to decompose it into two separate relations, using strategy 1 discussed above: P1(Ss#, Car_lic#) and P2(Ss#, Phone#).

15.3.5 Second Normal Form

Second normal form (2NF) is based on the concept of *full functional dependency*. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does *not* functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure 15.3(b), {Ssn, Pnumber} → Hours is a full dependency (neither Ssn → Hours nor Pnumber → Hours holds). However, the dependency {Ssn, Pnumber} → Ename is partial because Ssn → Ename holds.

> **Definition.** A relation schema R is in 2NF if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure 15.3(b) is in 1NF but is not in 2NF. The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3. The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.

If a relation schema is not in 2NF, it can be *second normalized* or *2NF normalized* into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure 15.3(b) lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 15.11(a), each of which is in 2NF.

15.3.6 Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R,[10] and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency Ssn → Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure 15.3(a), because both the

[10]This is the general definition of transitive dependency. Because we are concerned only with primary keys in this section, we allow transitive dependencies where X is the primary key but Z may be (a subset of) a candidate key.

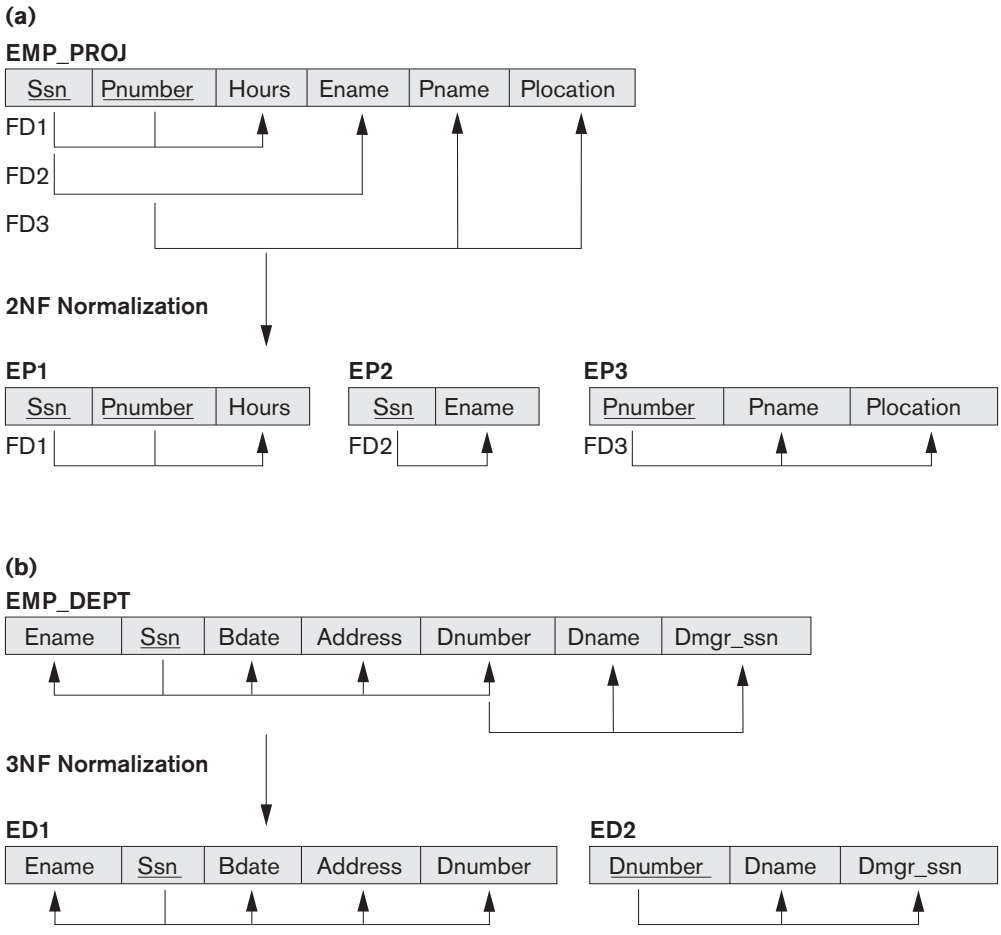


Figure 15.11
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

dependencies Ssn → Dnumber and Dnumber → Dmgr_ssn hold *and* Dnumber is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of Dmgr_ssn on Dnumber is undesirable in EMP_DEPT since Dnumber is not a key of EMP_DEPT.

> **Definition.** According to Codd's original definition, a relation schema R is in **3NF** if it satisfies 2NF *and* no nonprime attribute of R is transitively dependent on the primary key.

The relation schema EMP_DEPT in Figure 15.3(a) is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber. We can normalize

EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure 15.11(b). Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Intuitively, we can see that any functional dependency in which the left-hand side is part (a proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute, is a *problematic* FD. 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF. Table 15.1 informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding *remedy* or normalization performed to achieve the normal form.

15.4 General Definitions of Second and Third Normal Forms

In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies discussed in Section 15.1.2. The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on the *primary key.* The normalization procedure described so far is useful for analysis in practical situations for a given database where primary keys have already been defined. These definitions, however, do not take other candidate keys of a relation, if

Table 15.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF since it is independent of keys and functional dependencies. As a general definition of **prime attribute**, an attribute that is part of *any candidate key* will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.

15.4.1 General Definition of Second Normal Form

> **Definition.** A relation schema R is in **second normal form (2NF)** if every nonprime attribute A in R is not partially dependent on *any* key of R.[11]

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure 15.12(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.

Based on the two candidate keys Property_id# and {County_name, Lot#}, the functional dependencies FD1 and FD2 in Figure 15.12(a) hold. We choose Property_id# as the primary key, so it is underlined in Figure 15.12(a), but no special consideration will be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: County_name → Tax_rate
FD4: Area → Price

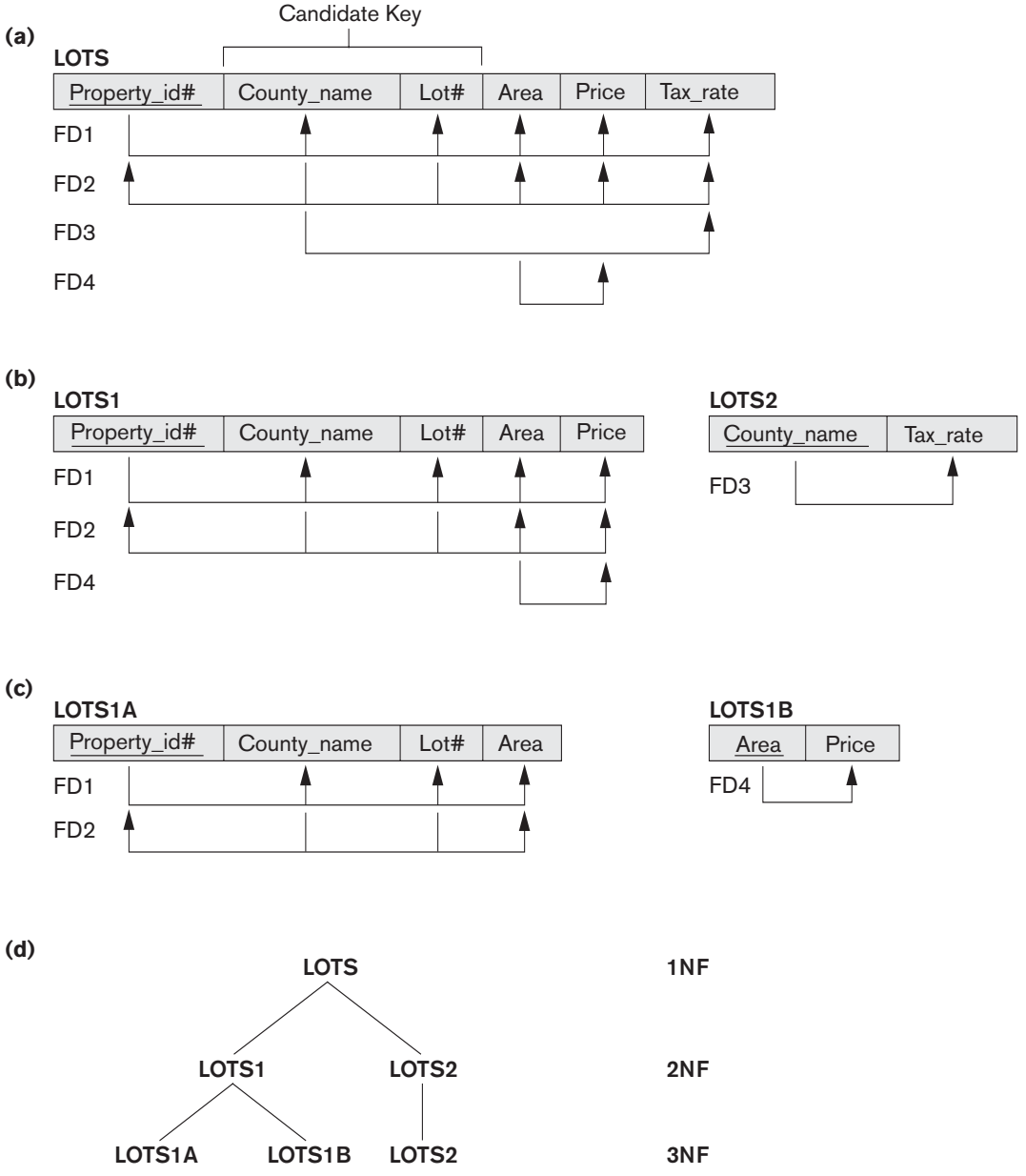
In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.)

The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key {County_name, Lot#}, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 15.12(b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

---

[11] This definition can be restated as follows: A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on *every* key of R.

Figure 15.12
Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.



15.4.2 General Definition of Third Normal Form

Definition. A relation schema R is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R.

According to this definition, LOTS2 (Figure 15.12(b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 15.12(c). We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the left-hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Two points are worth noting about this example and the general definition of 3NF:

- LOTS1 violates 3NF because Price is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute Area.
- This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF. Therefore, we could decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence, the transitive and partial dependencies that violate 3NF can be removed *in any order*.

15.4.3 Interpreting the General Definition of Third Normal Form

A relation schema R violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in R that does not meet either condition—meaning that it violates *both* conditions (a) and (b) of 3NF. This can occur due to two types of problematic functional dependencies:

- A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.
- A proper subset of a key of R functionally determines a nonprime attribute. Here we have a partial dependency that violates 3NF (and also 2NF).

Therefore, we can state a **general alternative definition of 3NF** as follows:

Alternative Definition. A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:

- It is fully functionally dependent on every key of R.
- It is nontransitively dependent on every key of R.

15.5 Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema in Figure 15.12(a) with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: DeKalb and Fulton. Suppose also that lot sizes in DeKalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, ..., 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: Area → County_name. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because County_name is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation *R*(Area, County_name), since there are only 16 possible Area values (see Figure 15.13). This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

Definition. A relation schema *R* is in **BCNF** if whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in *R*, then *X* is a superkey of *R*.

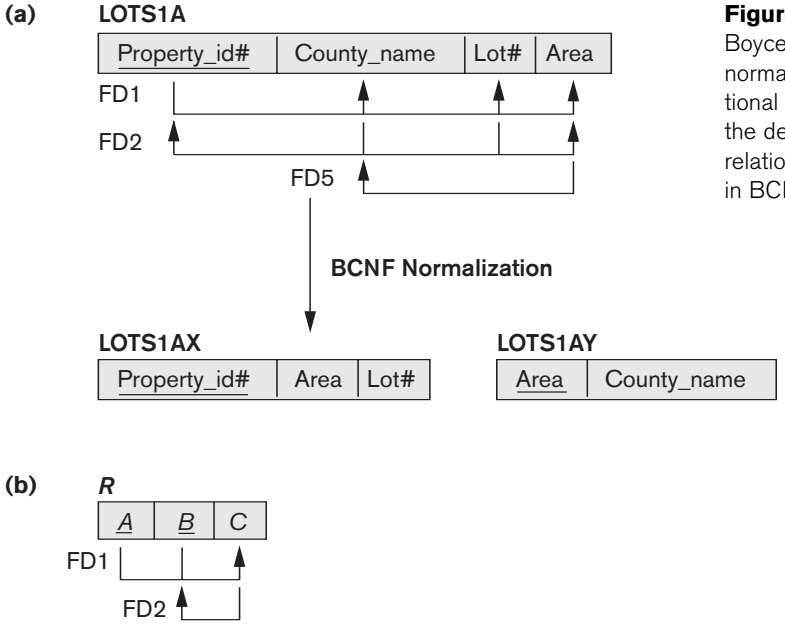


Figure 15.13
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

The formal definition of BCNF differs from the definition of 3NF in that condition (b) of 3NF, which allows A to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF. In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because County_name is a prime attribute (condition b), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 15.13(a). This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in a relation schema R with X not being a superkey *and* A being a prime attribute will R be in 3NF but not in BCNF. The relation schema R shown in Figure 15.13(b) illustrates the general case of such a relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization status of just 1NF or 2NF is not considered adequate, since they were developed historically as stepping stones to 3NF and BCNF.

As another example, consider Figure 15.14, which shows a relation TEACH with the following dependencies:

FD1: {Student, Course} → Instructor
FD2:[12] Instructor → Course

Note that {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 15.13(b), with Student as A, Course as B, and Instructor as C. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be

Figure 15.14
A relation TEACH that is in 3NF but not BCNF.

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

[12]This dependency means that *each instructor teaches one course* is a constraint for this application.

decomposed into one of the three following possible pairs:

1. {Student, Instructor} and {Student, Course}.
2. {Course, Instructor} and {Course, Student}.
3. {Instructor, Course} and {Instructor, Student}.

All three decompositions *lose* the functional dependency FD1. The *desirable decomposition* of those just shown is 3 because it will not generate spurious tuples after a join.

A test to determine whether a decomposition is nonadditive (or lossless) is discussed in Section 16.2.4 under Property NJB. In general, a relation not in BCNF should be decomposed so as to meet this property.

We make sure that we meet this property, because nonadditive decomposition is a must during normalization. We may have to possibly forgo the preservation of all functional dependencies in the decomposed relations, as is the case in this example. Algorithm 16.5 does that and could be used above to give decomposition 3 for TEACH, which yields two relations in BCNF as:

(Instructor, Course) and (Instructor, Student)

Note that if we designate (Student, Instructor) as a primary key of the relation TEACH, the FD Instructor → Course causes a partial (non-full-functional) dependency of Course on a part of this key. This FD may be removed as a part of second normalization yielding exactly the same two relations in the result. This is an example of a case where we may reach the same ultimate BCNF design via alternate paths of normalization.

15.6 Multivalued Dependency and Fourth Normal Form

So far we have discussed the concept of functional dependency, which is by far the most important type of dependency in relational database design theory, and normal forms based on functional dependencies. However, in many cases relations have constraints that cannot be specified as functional dependencies. In this section, we discuss the concept of *multivalued dependency* (MVD) and define *fourth normal form,* which is based on this dependency. A more formal discussion of MVDs and their properties is deferred to Chapter 16. Multivalued dependencies are a consequence of first normal form (1NF) (see Section 15.3.4), which disallows an attribute in a tuple to have a *set of values*, and the accompanying process of converting an unnormalized relation into 1NF. If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

15.2. Discuss insertion, deletion, and modification anomalies. Why are they considered bad? Illustrate with examples.

15.3. Why should NULLs in a relation be avoided as much as possible? Discuss the problem of spurious tuples and how we may prevent it.

15.4. State the informal guidelines for relation schema design that we discussed. Illustrate how violation of these guidelines may be harmful.

15.5. What is a functional dependency? What are the possible sources of the information that defines the functional dependencies that hold among the attributes of a relation schema?

15.6. Why can we not infer a functional dependency automatically from a particular relation state?

15.7. What does the term *unnormalized relation* refer to? How did the normal forms develop historically from first normal form up to Boyce-Codd normal form?

15.8. Define first, second, and third normal forms when only primary keys are considered. How do the general definitions of 2NF and 3NF, which consider all keys of a relation, differ from those that consider only primary keys?

15.9. What undesirable dependencies are avoided when a relation is in 2NF?

15.10. What undesirable dependencies are avoided when a relation is in 3NF?

15.11. In what way do the generalized definitions of 2NF and 3NF extend the definitions beyond primary keys?

15.12. Define Boyce-Codd normal form. How does it differ from 3NF? Why is it considered a stronger form of 3NF?

15.13. What is multivalued dependency? When does it arise?

15.14. Does a relation with two or more columns always have an MVD? Show with an example.

15.15. Define fourth normal form. When is it violated? When is it typically applicable?

15.16. Define join dependency and fifth normal form.

15.17. Why is 5NF also called project-join normal form (PJNF)?

15.18. Why do practical database designs typically aim for BCNF and not aim for higher normal forms?

Exercises

15.19. Suppose that we have the following requirements for a university database that is used to keep track of students' transcripts:

a. The university keeps track of each student's name (Sname), student number (Snum), Social Security number (Ssn), current address (Sc_addr) and

phone (Sc_phone), permanent address (Sp_addr) and phone (Sp_phone), birth date (Bdate), sex (Sex), class (Class) ('freshman', 'sophomore', ... , 'graduate'), major department (Major_code), minor department (Minor_code) (if any), and degree program (Prog) ('b.a.', 'b.s.', ... , 'ph.d.'). Both Ssn and student number have unique values for each student.

b. Each department is described by a name (Dname), department code (Dcode), office number (Doffice), office phone (Dphone), and college (Dcollege). Both name and code have unique values for each department.

c. Each course has a course name (Cname), description (Cdesc), course number (Cnum), number of semester hours (Credit), level (Level), and offering department (Cdept). The course number is unique for each course.

d. Each section has an instructor (Iname), semester (Semester), year (Year), course (Sec_course), and section number (Sec_num). The section number distinguishes different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ..., up to the total number of sections taught during each semester.

e. A grade record refers to a student (Ssn), a particular section, and a grade (Grade).

Design a relational database schema for this database application. First show all the functional dependencies that should hold among the attributes. Then design relation schemas for the database that are each in 3NF or BCNF. Specify the key attributes of each relation. Note any unspecified requirements, and make appropriate assumptions to render the specification complete.

15.20. What update anomalies occur in the EMP_PROJ and EMP_DEPT relations of Figures 15.3 and 15.4?

15.21. In what normal form is the LOTS relation schema in Figure 15.12(a) with respect to the restrictive interpretations of normal form that take *only the primary key* into account? Would it be in the same normal form if the general definitions of normal form were used?

15.22. Prove that any relation schema with two attributes is in BCNF.

15.23. Why do spurious tuples occur in the result of joining the EMP_PROJ1 and EMP_ LOCS relations in Figure 15.5 (result shown in Figure 15.6)?

15.24. Consider the universal relation $R = \{A, B, C, D, E, F, G, H, I, J\}$ and the set of functional dependencies $F = \{ \{A, B\} \rightarrow \{C\}, \{A\} \rightarrow \{D, E\}, \{B\} \rightarrow \{F\}, \{F\} \rightarrow \{G, H\}, \{D\} \rightarrow \{I, J\} \}$. What is the key for R? Decompose R into 2NF and then 3NF relations.

15.25. Repeat Exercise 15.24 for the following different set of functional dependencies $G = \{\{A, B\} \rightarrow \{C\}, \{B, D\} \rightarrow \{E, F\}, \{A, D\} \rightarrow \{G, H\}, \{A\} \rightarrow \{I\}, \{H\} \rightarrow \{J\} \}$.

15.26. Consider the following relation:

A	B	C	TUPLE#
10	b1	c1	1
10	b2	c2	2
11	b4	c1	3
12	b3	c4	4
13	b1	c1	5
14	b3	c4	6

a. Given the previous extension (state), which of the following dependencies *may hold* in the above relation? If the dependency cannot hold, explain why *by specifying the tuples that cause the violation*.

i. $A \rightarrow B$, ii. $B \rightarrow C$, iii. $C \rightarrow B$, iv. $B \rightarrow A$, v. $C \rightarrow A$

b. Does the above relation have a potential candidate key? If it does, what is it? If it does not, why not?

15.27. Consider a relation $R(A, B, C, D, E)$ with the following dependencies:

$AB \rightarrow C$, $CD \rightarrow E$, $DE \rightarrow B$

Is AB a candidate key of this relation? If not, is ABD? Explain your answer.

15.28. Consider the relation R, which has attributes that hold schedules of courses and sections at a university; R = {Course_no, Sec_no, Offering_dept, Credit_hours, Course_level, Instructor_ssn, Semester, Year, Days_hours, Room_no, No_of_students}. Suppose that the following functional dependencies hold on R:

{Course_no} → {Offering_dept, Credit_hours, Course_level}
{Course_no, Sec_no, Semester, Year} → {Days_hours, Room_no, No_of_students, Instructor_ssn}
{Room_no, Days_hours, Semester, Year} → {Instructor_ssn, Course_no, Sec_no}

Try to determine which sets of attributes form keys of R. How would you normalize this relation?

15.29. Consider the following relations for an order-processing application database at ABC, Inc.

ORDER (O#, Odate, Cust#, Total_amount)
ORDER_ITEM(O#, I#, Qty_ordered, Total_price, Discount%)

Assume that each item has a different discount. The Total_price refers to one item, Odate is the date on which the order was placed, and the Total_amount is the amount of the order. If we apply a natural join on the relations ORDER_ITEM and ORDER in this database, what does the resulting relation schema look like? What will be its key? Show the FDs in this resulting relation. Is it in 2NF? Is it in 3NF? Why or why not? (State assumptions, if you make any.)

15.30. Consider the following relation:

CAR_SALE(Car#, Date_sold, Salesperson#, Commission%, Discount_amt)

Assume that a car may be sold by multiple salespeople, and hence {Car#, Salesperson#} is the primary key. Additional dependencies are

Date_sold → Discount_amt and
Salesperson# → Commission%

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? How would you successively normalize it completely?

15.31. Consider the following relation for published books:

BOOK (Book_title, Author_name, Book_type, List_price, Author_affil, Publisher)

Author_affil refers to the affiliation of author. Suppose the following dependencies exist:

Book_title → Publisher, Book_type
Book_type → List_price
Author_name → Author_affil

a. What normal form is the relation in? Explain your answer.

b. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

15.32. This exercise asks you to convert business statements into dependencies. Consider the relation DISK_DRIVE (Serial_number, Manufacturer, Model, Batch, Capacity, Retailer). Each tuple in the relation DISK_DRIVE contains information about a disk drive with a unique Serial_number, made by a manufacturer, with a particular model number, released in a certain batch, which has a certain storage capacity and is sold by a certain retailer. For example, the tuple Disk_drive ('1978619', 'WesternDigital', 'A2235X', '765234', 500, 'CompUSA') specifies that WesternDigital made a disk drive with serial number 1978619 and model number A2235X, released in batch 765234; it is 500GB and sold by CompUSA.

Write each of the following dependencies as an FD:

a. The manufacturer and serial number uniquely identifies the drive.

b. A model number is registered by a manufacturer and therefore can't be used by another manufacturer.

c. All disk drives in a particular batch are the same model.

d. All disk drives of a certain model of a particular manufacturer have exactly the same capacity.

15.33. Consider the following relation:

R (Doctor#, Patient#, Date, Diagnosis, Treat_code, Charge)

In the above relation, a tuple describes a visit of a patient to a doctor along with a treatment code and daily charge. Assume that diagnosis is determined (uniquely) for each patient by a doctor. Assume that each treatment code has a fixed charge (regardless of patient). Is this relation in 2NF? Justify your answer and decompose if necessary. Then argue whether further normalization to 3NF is necessary, and if so, perform it.

15.34. Consider the following relation:

CAR_SALE (Car_id, Option_type, Option_listprice, Sale_date, Option_discountedprice)

This relation refers to options installed in cars (e.g., cruise control) that were sold at a dealership, and the list and discounted prices of the options.

If CarID $\rightarrow$ Sale_date and Option_type $\rightarrow$ Option_listprice and CarID, Option_type $\rightarrow$ Option_discountedprice, argue using the generalized definition of the 3NF that this relation is not in 3NF. Then argue from your knowledge of 2NF, why it is not even in 2NF.

15.35. Consider the relation:

BOOK (Book_Name, Author, Edition, Year)

with the data:

Book_Name	Author	Edition	Copyright_Year
DB_fundamentals	Navathe	4	2004
DB_fundamentals	Elmasri	4	2004
DB_fundamentals	Elmasri	5	2007
DB_fundamentals	Navathe	5	2007

a. Based on a common-sense understanding of the above data, what are the possible candidate keys of this relation?

b. Justify that this relation has the MVD { Book } $\twoheadrightarrow$ { Author } | { Edition, Year }.

c. What would be the decomposition of this relation based on the above MVD? Evaluate each resulting relation for the highest normal form it possesses.

15.36. Consider the following relation:

TRIP (Trip_id, Start_date, Cities_visited, Cards_used)

This relation refers to business trips made by company salespeople. Suppose the TRIP has a single Start_date, but involves many Cities and salespeople may use multiple credit cards on the trip. Make up a mock-up population of the table.

a. Discuss what FDs and/or MVDs exist in this relation.

b. Show how you will go about normalizing it.

Laboratory Exercise

Note: The following exercise use the DBD (Data Base Designer) system that is described in the laboratory manual. The relational schema R and set of functional dependencies F need to be coded as lists. As an example, R and F for this problem is coded as:

$R = [a, b, c, d, e, f, g, h, i, j]$
$F = [[[a, b],[c]],$
$[[a],[d, e]],$
$[[b],[f]],$
$[[f],[g, h]],$
$[[d],[i, j]]]$

Since DBD is implemented in Prolog, use of uppercase terms is reserved for variables in the language and therefore lowercase constants are used to code the attributes. For further details on using the DBD system, please refer to the laboratory manual.

15.37. Using the DBD system, verify your answers to the following exercises:

a. 15.24 (3NF only)

b. 15.25

c. 15.27

d. 15.28

Selected Bibliography

Functional dependencies were originally introduced by Codd (1970). The original definitions of first, second, and third normal form were also defined in Codd (1972a), where a discussion on update anomalies can be found. Boyce-Codd normal form was defined in Codd (1974). The alternative definition of third normal form is given in Ullman (1988), as is the definition of BCNF that we give here. Ullman (1988), Maier (1983), and Atzeni and De Antonellis (1993) contain many of the theorems and proofs concerning functional dependencies.

Additional references to relational design theory are given in Chapter 16.

- Since B → A, by augmenting with *B* on both sides (IR2), we have *BB* → *AB*, or *B* → *AB* (i). However, *AB* → *D* as given (ii).
- Hence by the transitive rule (IR3), we get from (i) and (ii), *B* → *D*. Thus *AB* → *D* may be replaced by *B* → *D*.
- We now have a set equivalent to original *E*, say *E′*: {*B* → *A*, *D* → *A*, *B* → *D*}. No further reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.
- In step 3 we look for a redundant FD in *E′*. By using the transitive rule on *B* → *D* and *D* → *A*, we derive *B* → *A*. Hence *B* → *A* is redundant in *E′* and can be eliminated.
- Therefore, the minimal cover of *E* is {*B* → *D*, *D* → *A*}.

In Section 16.3 we will see how relations can be synthesized from a given set of dependencies *E* by first finding the minimal cover *F* for *E*.

Next, we provide a simple algorithm to determine the key of a relation:

Algorithm 16.2(a). Finding a Key *K* for *R* Given a set *F* of Functional Dependencies

Input: A relation *R* and a set of functional dependencies *F* on the attributes of *R*.

1. Set *K* := *R*.
2. For each attribute *A* in *K*

   {compute $(K - A)^+$ with respect to *F*;

   if $(K - A)^+$ contains all the attributes in R, then set K := K – {A} };

In Algoritm 16.2(a), we start by setting *K* to all the attributes of *R*; we then remove one attribute at a time and check whether the remaining attributes still form a superkey. Notice, too, that Algorithm 16.2(a) determines only *one key* out of the possible candidate keys for *R*; the key returned depends on the order in which attributes are removed from *R* in step 2.

16.2 Properties of Relational Decompositions

We now turn our attention to the process of decomposition that we used throughout Chapter 15 to decompose relations in order to get rid of unwanted dependencies and achieve higher normal forms. In Section 16.2.1 we give examples to show that looking at an *individual* relation to test whether it is in a higher normal form does not, on its own, guarantee a good design; rather, a *set of relations* that together form the relational database schema must possess certain additional properties to ensure a good design. In Sections 16.2.2 and 16.2.3 we discuss two of these properties: the dependency preservation property and the nonadditive (or lossless) join property. Section 16.2.4 discusses binary decompositions and Section 16.2.5 discusses successive nonadditive join decompositions.

16.2.1 Relation Decomposition and Insufficiency of Normal Forms

The relational database design algorithms that we present in Section 16.3 start from a single **universal relation schema** $R = \{A_1, A_2, ..., A_n\}$ that includes *all* the attributes of the database. We implicitly make the **universal relation assumption**, which states that every attribute name is unique. The set F of functional dependencies that should hold on the attributes of R is specified by the database designers and is made available to the design algorithms. Using the functional dependencies, the algorithms decompose the universal relation schema R into a set of relation schemas $D = \{R_1, R_2, ..., R_m\}$ that will become the relational database schema; D is called a **decomposition** of R.

We must make sure that each attribute in R will appear in at least one relation schema R_i in the decomposition so that no attributes are *lost*; formally, we have

$$\bigcup_{i=1}^{m} R_i = R$$

This is called the **attribute preservation** condition of a decomposition.

Another goal is to have each individual relation R_i in the decomposition D be in BCNF or 3NF. However, this condition is not sufficient to guarantee a good database design on its own. We must consider the decomposition of the universal relation as a whole, in addition to looking at the individual relations. To illustrate this point, consider the EMP_LOCS(Ename, Plocation) relation in Figure 15.5, which is in 3NF and also in BCNF. In fact, any relation schema with only two attributes is automatically in BCNF.[5] Although EMP_LOCS is in BCNF, it still gives rise to spurious tuples when joined with EMP_PROJ (Ssn, Pnumber, Hours, Pname, Plocation), which is not in BCNF (see the result of the natural join in Figure 15.6). Hence, EMP_LOCS represents a particularly bad relation schema because of its convoluted semantics by which Plocation gives the location of *one of the projects* on which an employee works. Joining EMP_LOCS with PROJECT(Pname, Pnumber, Plocation, Dnum) in Figure 15.2—which *is* in BCNF—using Plocation as a joining attribute also gives rise to spurious tuples. This underscores the need for other criteria that, together with the conditions of 3NF or BCNF, prevent such bad designs. In the next three subsections we discuss such additional conditions that should hold on a decomposition D as a whole.

16.2.2 Dependency Preservation Property of a Decomposition

It would be useful if each functional dependency $X \rightarrow Y$ specified in F either appeared directly in one of the relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i. Informally, this is the *dependency preservation condition*. We want to preserve the dependencies because

[5]As an exercise, the reader should prove that this statement is true.

each dependency in F represents a constraint on the database. If one of the dependencies is not represented in some individual relation R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation. We may have to join multiple relations so as to include all attributes involved in that dependency.

It is not necessary that the exact dependencies specified in F appear themselves in individual relations of the decomposition D. It is sufficient that the union of the dependencies that hold on the individual relations in D be equivalent to F. We now define these concepts more formally.

> **Definition.** Given a set of dependencies F on R, the **projection** of F on R_i, denoted by $\pi_{Ri}(F)$ where R_i is a subset of R, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i. Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all their left- and right-hand-side attributes are in R_i. We say that a decomposition $D = \{R_1, R_2, ..., R_m\}$ of R is **dependency-preserving** with respect to F if the union of the projections of F on each R_i in D is equivalent to F; that is, $((\pi_{R_1}(F)) \cup ... \cup (\pi_{R_m}(F)))^+ = F^+$.

If a decomposition is not dependency-preserving, some dependency is **lost** in the decomposition. To check that a lost dependency holds, we must take the JOIN of two or more relations in the decomposition to get a relation that includes all left- and right-hand-side attributes of the lost dependency, and then check that the dependency holds on the result of the JOIN—an option that is not practical.

An example of a decomposition that does not preserve dependencies is shown in Figure 15.13(a), in which the functional dependency FD2 is lost when LOTS1A is decomposed into {LOTS1AX, LOTS1AY}. The decompositions in Figure 15.12, however, are dependency-preserving. Similarly, for the example in Figure 15.14, no matter what decomposition is chosen for the relation TEACH(Student, Course, Instructor) from the three provided in the text, one or both of the dependencies originally present are bound to be lost. We state a claim below related to this property without providing any proof.

> **Claim 1.** It is always possible to find a dependency-preserving decomposition D with respect to F such that each relation R_i in D is in 3NF.

In Section 16.3.1, we describe Algorithm 16.4, which creates a dependency-preserving decomposition $D = \{R_1, R_2, ..., R_m\}$ of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in 3NF.

16.2.3 Nonadditive (Lossless) Join Property of a Decomposition

Another property that a decomposition D should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition. We already illustrated this problem in Section 15.1.4 with the example in Figures 15.5

and 15.6. Because this is a property of a decomposition of relation *schemas,* the condition of no spurious tuples should hold on *every legal relation state*—that is, every relation state that satisfies the functional dependencies in *F*. Hence, the lossless join property is always defined with respect to a specific set *F* of dependencies.

> **Definition.** Formally, a decomposition $D = \{R_1, R_2, ..., R_m\}$ of R has the **lossless (nonadditive) join property** with respect to the set of dependencies F on R if, for *every* relation state r of R that satisfies F, the following holds, where $*$ is the NATURAL JOIN of all the relations in D: $*(\pi_{R_1}(r), ..., \pi_{R_m}(r)) = r$.

The word loss in *lossless* refers to *loss of information*, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT (π) and NATURAL JOIN ($*$) operations are applied; these additional tuples represent erroneous or invalid information. We prefer the term *nonadditive join* because it describes the situation more accurately. Although the term *lossless join* has been popular in the literature, *we will henceforth use the term nonadditive join*, which is self-explanatory and unambiguous. The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations. We may, however, sometimes use the term **lossy design** to refer to a design that represents a loss of information (see example at the end of Algorithm 16.4).

The decomposition of EMP_PROJ(Ssn, Pnumber, Hours, Ename, Pname, Plocation) in Figure 15.3 into EMP_LOCS(Ename, Plocation) and EMP_PROJ1(Ssn, Pnumber, Hours, Pname, Plocation) in Figure 15.5 obviously does not have the nonadditive join property, as illustrated by Figure 15.6. We will use a general procedure for testing whether any decomposition D of a relation into n relations is nonadditive with respect to a set of given functional dependencies F in the relation; it is presented as Algorithm 16.3 below. It is possible to apply a simpler test to check if the decomposition is nonadditive for binary decompositions; that test is described in Section 16.2.4.

Algorithm 16.3. Testing for Nonadditive Join Property

Input: A universal relation R, a decomposition $D = \{R_1, R_2, ..., R_m\}$ of R, and a set F of functional dependencies.

Note: Explanatory comments are given at the end of some of the steps. They follow the format: (* *comment* *).

1. Create an initial matrix S with one row i for each relation R_i in D, and one column j for each attribute A_j in R.
2. Set $S(i, j) := b_{ij}$ for all matrix entries. (* each b_{ij} is a distinct symbol associated with indices (i, j) *).
3. For each row i representing relation schema R_i
   {for each column j representing attribute A_j
   {if (relation R_i includes attribute A_j) then set $S(i, j) := a_j$;};}; (* each a_j is a distinct symbol associated with index (j) *).

4. Repeat the following loop until a *complete loop execution* results in no changes to S
   {for each functional dependency $X \rightarrow Y$ in F
      {for all rows in S *that have the same symbols* in the columns corresponding to attributes in X
         {make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: If any of the rows has an a symbol for the column, set the other rows to that *same a* symbol in the column. If no a symbol exists for the attribute in any of the rows, choose one of the b symbols that appears in one of the rows for the attribute and set the other rows to that same b symbol in the column ;} ; } ;};
5. If a row is made up entirely of a symbols, then the decomposition has the nonadditive join property; otherwise, it does not.

Given a relation R that is decomposed into a number of relations R_1, R_2, ..., R_m, Algorithm 16.3 begins the matrix S that we consider to be some relation state r of R. Row i in S represents a tuple t_i (corresponding to relation R_i) that has a symbols in the columns that correspond to the attributes of R_i and b symbols in the remaining columns. The algorithm then transforms the rows of this matrix (during the loop in step 4) so that they represent tuples that satisfy all the functional dependencies in F. At the end of step 4, any two rows in S—which represent two tuples in r—that agree in their values for the left-hand-side attributes X of a functional dependency $X \rightarrow Y$ in F will also agree in their values for the right-hand-side attributes Y. It can be shown that after applying the loop of step 4, if any row in S ends up with all a symbols, then the decomposition D has the nonadditive join property with respect to F.

If, on the other hand, no row ends up being all a symbols, D does not satisfy the lossless join property. In this case, the relation state r represented by S at the end of the algorithm will be an example of a relation state r of R that satisfies the dependencies in F but does not satisfy the nonadditive join condition. Thus, this relation serves as a **counterexample** that proves that D does not have the nonadditive join property with respect to F. Note that the a and b symbols have no special meaning at the end of the algorithm.

Figure 16.1(a) shows how we apply Algorithm 16.3 to the decomposition of the EMP_PROJ relation schema from Figure 15.3(b) into the two relation schemas EMP_PROJ1 and EMP_LOCS in Figure 15.5(a). The loop in step 4 of the algorithm cannot change any b symbols to a symbols; hence, the resulting matrix S does not have a row with all a symbols, and so the decomposition does not have the nonadditive join property.

Figure 16.1(b) shows another decomposition of EMP_PROJ (into EMP, PROJECT, and WORKS_ON) that does have the nonadditive join property, and Figure 16.1(c) shows how we apply the algorithm to that decomposition. Once a row consists only of a symbols, we conclude that the decomposition has the nonadditive join property, and we can stop applying the functional dependencies (step 4 in the algorithm) to the matrix S.

Figure 16.1
Nonadditive join test for *n*-ary decompositions. (a) Case 1: Decomposition of EMP_PROJ into EMP_PROJ1 and EMP_LOCS fails test. (b) A decomposition of EMP_PROJ that has the lossless join property. (c) Case 2: Decomposition of EMP_PROJ into EMP, PROJECT, and WORKS_ON satisfies test.

(a) R = {Ssn, Ename, Pnumber, Pname, Plocation, Hours} $D = \{R_1, R_2\}$
R_1 = EMP_LOCS = {Ename, Plocation}
R_2 = EMP_PROJ1 = {Ssn, Pnumber, Hours, Pname, Plocation}

F = {Ssn → Ename; Pnumber → {Pname, Plocation}; {Ssn, Pnumber} → Hours}

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R_2	a_1	b_{22}	a_3	a_4	a_5	a_6

(No changes to matrix after applying functional dependencies)

(b)

EMP

Ssn	Ename

PROJECT

Pnumber	Pname	Plocation

WORKS_ON

Ssn	Pnumber	Hours

(c) R = {Ssn, Ename, Pnumber, Pname, Plocation, Hours} $D = \{R_1, R_2, R_3\}$
R_1 = EMP = {Ssn, Ename}
R_2 = PROJ = {Pnumber, Pname, Plocation}
R_3 = WORKS_ON = {Ssn, Pnumber, Hours}

F = {Ssn → Ename; Pnumber → {Pname, Plocation}; {Ssn, Pnumber} → Hours}

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

(Original matrix S at start of algorithm)

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	~~b_{32}~~ a_2	a_3	~~b_{34}~~ a_4	~~b_{35}~~ a_5	a_6

(Matrix S after applying the first two functional dependencies; last row is all "a" symbols so we stop)

16.2.4 Testing Binary Decompositions for the Nonadditive Join Property

Algorithm 16.3 allows us to test whether a particular decomposition D into n relations obeys the nonadditive join property with respect to a set of functional dependencies F. There is a special case of a decomposition called a **binary decomposition**—decomposition of a relation R into two relations. We give an easier test to apply than Algorithm 16.3, but while it is very handy to use, it is *limited* to binary decompositions only.

> **Property NJB (Nonadditive Join Test for Binary Decompositions).** A decomposition $D = \{R_1, R_2\}$ of R has the lossless (nonadditive) join property with respect to a set of functional dependencies F on R *if and only if* either
>
> - The FD $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ is in F^+, or
> - The FD $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ is in F^+

You should verify that this property holds with respect to our informal successive normalization examples in Sections 15.3 and 15.4. In Section 15.5 we decomposed LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, and decomposed the TEACH relation in Figure 15.14 into the two relations {Instructor, Course} and {Instructor, Student}. These are valid decompositions because they are nonadditive per the above test.

16.2.5 Successive Nonadditive Join Decompositions

We saw the successive decomposition of relations during the process of second and third normalization in Sections 15.3 and 15.4. To verify that these decompositions are nonadditive, we need to ensure another property, as set forth in Claim 2.

> **Claim 2 (Preservation of Nonadditivity in Successive Decompositions).** If a decomposition $D = \{R_1, R_2, ..., R_m\}$ of R has the nonadditive (lossless) join property with respect to a set of functional dependencies F on R, and if a decomposition $D_i = \{Q_1, Q_2, ..., Q_k\}$ of R_i has the nonadditive join property with respect to the projection of F on R_i, then the decomposition $D_2 = \{R_1, R_2, ..., R_{i-1}, Q_1, Q_2, ..., Q_k, R_{i+1}, ..., R_m\}$ of R has the nonadditive join property with respect to F.

16.3 Algorithms for Relational Database Schema Design

We now give three algorithms for creating a relational decomposition from a universal relation. Each algorithm has specific properties, as we discuss next.

arbitrary types of constraints. We pointed out the need for arithmetic functions or more complex procedures to enforce certain functional dependency constraints. We concluded with a brief discussion of the domain-key normal form (DKNF).

Review Questions

16.1. What is the role of Armstrong's inference rules (inference rules IR1 through IR3) in the development of the theory of relational design?

16.2. What is meant by the completeness and soundness of Armstrong's inference rules?

16.3. What is meant by the closure of a set of functional dependencies? Illustrate with an example.

16.4. When are two sets of functional dependencies equivalent? How can we determine their equivalence?

16.5. What is a minimal set of functional dependencies? Does every set of dependencies have a minimal equivalent set? Is it always unique?

16.6. What is meant by the attribute preservation condition on a decomposition?

16.7. Why are normal forms alone insufficient as a condition for a good schema design?

16.8. What is the dependency preservation property for a decomposition? Why is it important?

16.9. Why can we not guarantee that BCNF relation schemas will be produced by dependency-preserving decompositions of non-BCNF relation schemas? Give a counterexample to illustrate this point.

16.10. What is the lossless (or nonadditive) join property of a decomposition? Why is it important?

16.11. Between the properties of dependency preservation and losslessness, which one must definitely be satisfied? Why?

16.12. Discuss the NULL value and dangling tuple problems.

16.13. Illustrate how the process of creating first normal form relations may lead to multivalued dependencies. How should the first normalization be done properly so that MVDs are avoided?

16.14. What types of constraints are inclusion dependencies meant to represent?

16.15. How do template dependencies differ from the other types of dependencies we discussed?

16.16. Why is the domain-key normal form (DKNF) known as the ultimate normal form?

Exercises

16.17. Show that the relation schemas produced by Algorithm 16.4 are in 3NF.

16.18. Show that, if the matrix S resulting from Algorithm 16.3 does not have a row that is all a symbols, projecting S on the decomposition and joining it back will always produce at least one spurious tuple.

16.19. Show that the relation schemas produced by Algorithm 16.5 are in BCNF.

16.20. Show that the relation schemas produced by Algorithm 16.6 are in 3NF.

16.21. Specify a template dependency for join dependencies.

16.22. Specify all the inclusion dependencies for the relational schema in Figure 3.5.

16.23. Prove that a functional dependency satisfies the formal definition of multivalued dependency.

16.24. Consider the example of normalizing the LOTS relation in Sections 15.4 and 15.5. Determine whether the decomposition of LOTS into {LOTS1AX, LOTS1AY, LOTS1B, LOTS2} has the lossless join property, by applying Algorithm 16.3 and also by using the test under Property NJB.

16.25. Show how the MVDs Ename $\twoheadrightarrow$ Pname and Ename $\twoheadrightarrow$ Dname in Figure 15.15(a) may arise during normalization into 1NF of a relation, where the attributes Pname and Dname are multivalued.

16.26. Apply Algorithm 16.2(a) to the relation in Exercise 15.24 to determine a key for R. Create a minimal set of dependencies G that is equivalent to F, and apply the synthesis algorithm (Algorithm 16.6) to decompose R into 3NF relations.

16.27. Repeat Exercise 16.26 for the functional dependencies in Exercise 15.25.

16.28. Apply the decomposition algorithm (Algorithm 16.5) to the relation R and the set of dependencies F in Exercise 15.24. Repeat for the dependencies G in Exercise 15.25.

16.29. Apply Algorithm 16.2(a) to the relations in Exercises 15.27 and 15.28 to determine a key for R. Apply the synthesis algorithm (Algorithm 16.6) to decompose R into 3NF relations and the decomposition algorithm (Algorithm 16.5) to decompose R into BCNF relations.

16.30. Write programs that implement Algorithms 16.5 and 16.6.

16.31. Consider the following decompositions for the relation schema R of Exercise 15.24. Determine whether each decomposition has (1) the dependency preservation property, and (2) the lossless join property, with respect to F. Also determine which normal form each relation in the decomposition is in.

a. $D_1 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C\}$, $R_2 = \{A, D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$

b. $D_2 = \{R_1, R_2, R_3\}$; $R_1 = \{A, B, C, D, E\}$, $R_2 = \{B, F, G, H\}$, $R_3 = \{D, I, J\}$

c. $D_3 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C, D\}$, $R_2 = \{D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$

16.32. Consider the relation REFRIG(Model#, Year, Price, Manuf_plant, Color), which is abbreviated as REFRIG(M, Y, P, MP, C), and the following set F of functional dependencies: $F = \{M \rightarrow MP, \{M, Y\} \rightarrow P, MP \rightarrow C\}$

a. Evaluate each of the following as a candidate key for REFRIG, giving reasons why it can or cannot be a key: {M}, {M, Y}, {M, C}.

b. Based on the above key determination, state whether the relation REFRIG is in 3NF and in BCNF, giving proper reasons.

c. Consider the decomposition of REFRIG into $D = \{R_1(M, Y, P), R_2(M, MP, C)\}$. Is this decomposition lossless? Show why. (You may consult the test under Property NJB in Section 16.2.4.)

Laboratory Exercises

Note: These exercises use the DBD (Data Base Designer) system that is described in the laboratory manual. The relational schema R and set of functional dependencies F need to be coded as lists. As an example, R and F for problem 15.24 are coded as:

$R = [a, b, c, d, e, f, g, h, i, j]$
$F = [[[a, b],[c]],$
$[[a],[d, e]],$
$[[b],[f]],$
$[[f],[g, h]],$
$[[d],[i, j]]]$

Since DBD is implemented in Prolog, use of uppercase terms is reserved for variables in the language and therefore lowercase constants are used to code the attributes. For further details on using the DBD system, please refer to the laboratory manual.

16.33. Using the DBD system, verify your answers to the following exercises:

a. 16.24

b. 16.26

c. 16.27

d. 16.28

e. 16.29

f. 16.31 (a) and (b)

g. 16.32 (a) and (c)