Part 2

Understanding Relational Databases

The chapters in Part 2 provide a detailed introduction to the Relational Data Model to instill a foundation for database design and application development with relational databases. Chapter 3 presents data definition concepts and retrieval operators for relational databases. Chapter 4 demonstrates SQL retrieval and modification statements for problems of basic and intermediate complexity and emphasizes mental tools to develop query formulation skills.

Chapter 3

The Relational Data Model

Learning Objectives

This chapter provides the foundation for using relational databases. After this chapter the student should have acquired the following knowledge and skills:

- Recognize relational database terminology.
- Understand the meaning of the integrity rules for relational databases.
- Understand the impact of referenced rows on maintaining relational databases.
- Understand the meaning of each relational algebra operator.
- List tables that must be combined to obtain desired results for simple retrieval requests.

Overview

The chapters in Part 1 provided a starting point for your exploration of database technology and your understanding of the database development process. You broadly learned about database characteristics, DBMS features, the goals of database development, and the phases of the database development process. This chapter narrows your focus to the relational data model. Relational DBMSs dominate the market for business DBMSs. You will undoubtedly use relational DBMSs throughout your career as an information systems professional. This chapter provides background so that you may become proficient in designing databases and developing applications for relational databases in later chapters.

To effectively use a relational database, you need two kinds of knowledge. First, you need to understand the structure and contents of the database tables. Understanding the connections among tables is especially critical because many database retrievals involve multiple tables. To help you understand relational databases, this chapter presents the basic terminology, the integrity rules, and a notation to visualize connections among tables. Second, you need to understand the operators of relational algebra as they are the building blocks of most commercial query languages. Understanding the operators will improve your knowledge of query languages such as SQL. To help you understand the meaning of each operator, this chapter provides a visual representation of each operator and several convenient summaries.

3.1 Basic Elements

Relational database systems were originally developed because of familiarity and simplicity. Because tables are used to communicate ideas in many fields, the terminology of tables, rows, and columns is not intimidating to most users. During the early years of relational databases (1970s), the simplicity and familiarity of relational databases had strong appeal especially as compared to the procedural orientation of other data models that existed at the time. Despite the familiarity and simplicity of relational databases, there is a strong mathematical basis also. The mathematics of relational databases involves conceptualizing tables as sets. The combination of familiarity and simplicity with a mathematical foundation is so powerful that relational DBMSs are commercially dominant.

This section presents the basic terminology of relational databases and introduces the CREATE TABLE statement of the Structured Query Language (SQL). Sections 3.2 through 3.4 provide more detail on the elements defined in this section.

3.1.1 Tables

table
a two-dimensional arrangement of data. A table consists of a heading defining the table name and column names and a body containing rows of data.

A relational database consists of a collection of tables. Each table has a heading or definition part and a body or content part. The heading part consists of the table name and the column names. For example, a student table may have columns for Social Security number, name, street address, city, state, zip, class (freshman, sophomore, etc.), major, and cumulative grade point average (GPA). The body shows the rows of the table. Each row in a student table represents a student enrolled at a university. A student table for a major university may have more than 30,000 rows, too many to view at one time.

To understand a table, it is also useful to view some of its rows. A table listing or datasheet shows the column names in the first row and the body in the other rows. Table 3.1 shows a table listing for the *Student* table. Three sample rows representing university students are displayed. In this book, the naming convention for column names uses a table abbreviation (Std) followed by a descriptive name. Because column names are often used without identifying the associated tables, the abbreviation supports easy table association. Mixed case highlights the different parts of a column name.

data type
defines a set of values and permissible operations on the values. Each column of a table is associated with a data type.

A CREATE TABLE statement can be used to define the heading part of a table. CREATE TABLE is a statement in the Structured Query Language (SQL). Because SQL is an industry standard language, the CREATE TABLE statement can be used to create tables in most DBMSs. The CREATE TABLE statement that follows creates the *Student* table.[1] For each column, the column name and data type are specified. Data types indicate the kind of data (character, numeric, Yes/No, etc.) and permissible operations (numeric operations, string operations etc.) for the column. Each data type has a name (for example,

TABLE 3.1 **Sample Table Listing of the Student Table**

StdSSN	StdFirstName	StdLastName	StdCity	StdState	StdZip	StdMajor	StdClass	StdGPA
123-45-6789	HOMER	WELLS	SEATTLE	WA	98121-1111	IS	FR	3.00
124-56-7890	BOB	NORBERT	BOTHELL	WA	98011-2121	FIN	JR	2.70
234-56-7890	CANDY	KENDALL	TACOMA	WA	99042-3321	ACCT	JR	3.50

[1] The CREATE TABLE statements in this chapter conform to the standard SQL syntax. There are slight syntax differences for most commercial DBMSs.

TABLE 3.2
Brief Description of Common SQL Data Types

Data Type	Description
CHAR(L)	For fixed-length text entries such as state abbreviations and Social Security numbers. Each column value using CHAR contains the maximum number of characters (*L*) even if the actual length is shorter. Most DBMSs have an upper limit on the length (*L*) such as 255.
VARCHAR(L)	For variable-length text such as names and street addresses. Column values using VARCHAR contain only the actual number of characters, not the maximum length for CHAR columns. Most DBMSs have an upper limit on the length such as 255.
FLOAT(P)	For columns containing numeric data with a floating precision such as interest rate calculations and scientific calculations. The precision parameter *P* indicates the number of significant digits. Most DBMSs have an upper limit on *P* such as 38. Some DBMSs have two data types, REAL and DOUBLE PRECISION, for low- and high-precision floating point numbers instead of the variable precision with the FLOAT data type.
DATE/TIME	For columns containing dates and times such as an order date. These data types are not standard across DBMSs. Some systems support three data types (DATE, TIME, and TIMESTAMP) while other systems support a combined data type (DATE) storing both the date and time.
DECIMAL(W, R)	For columns containing numeric data with a fixed precision such as monetary amounts. The *W* value indicates the total number of digits and the *R* value indicates the number of digits to the right of the decimal point. This data type is also called NUMERIC in some systems.
INTEGER	For columns containing whole numbers (i.e., numbers without a decimal point). Some DBMSs have the SMALLINT data type for very small whole numbers and the LONG data type for very large integers.
BOOLEAN	For columns containing data with two values such as true/false or yes/no.

CHAR for character) and usually a length specification. Table 3.2 lists common data types used in relational DBMSs.[2]

```
CREATE TABLE Student
(       StdSSN          CHAR(11),
        StdFirstName    VARCHAR(50),
        StdLastName     VARCHAR(50),
        StdCity         VARCHAR(50),
        StdState        CHAR(2),
        StdZip          CHAR(10),
        StdMajor        CHAR(6),
        StdClass        CHAR(6),
        StdGPA          DECIMAL(3,2)    )
```

relationship
connection between rows in two tables. Relationships are shown by column values in one table that match column values in another table.

3.1.2 Connections among Tables

It is not enough to understand each table individually. To understand a relational database, connections or relationships among tables also must be understood. The rows in a table are usually related to rows in other tables. Matching (identical) values indicate relationships between tables. Consider the sample *Enrollment* table (Table 3.3) in which each row represents a student enrolled in an offering of a course. The values in the *StdSSN* column of the *Enrollment* table match the *StdSSN* values in the sample *Student* table (Table 3.1). For

[2] Data types are not standard across relational DBMSs. The data types used in this chapter are specified in the latest SQL standard. Most DBMSs support these data types although the data type names may differ.

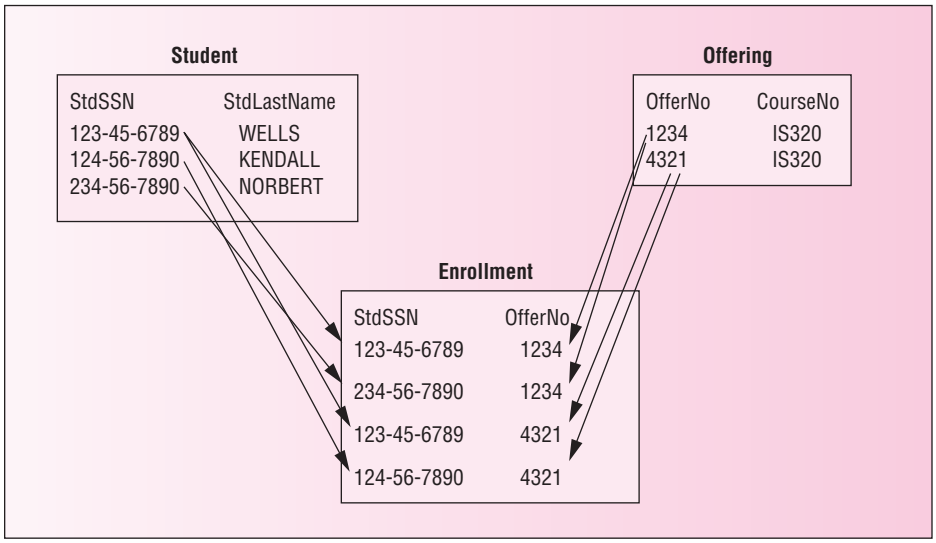
TABLE 3.3
Sample Enrollment Table

OfferNo	StdSSN	EnrGrade
1234	123-45-6789	3.3
1234	234-56-7890	3.5
4321	123-45-6789	3.5
4321	124-56-7890	3.2

TABLE 3.4 **Sample Offering Table**

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	FacSSN	OffDays
1111	IS320	SUMMER	2006	BLM302	10:30 AM		MW
1234	IS320	FALL	2005	BLM302	10:30 AM	098-76-5432	MW
2222	IS460	SUMMER	2005	BLM412	1:30 PM		TTH
3333	IS320	SPRING	2006	BLM214	8:30 AM	098-76-5432	MW
4321	IS320	FALL	2005	BLM214	3:30 PM	098-76-5432	TTH
4444	IS320	SPRING	2006	BLM302	3:30 PM	543-21-0987	TTH
5678	IS480	SPRING	2006	BLM302	10:30 AM	987-65-4321	MW
5679	IS480	SPRING	2006	BLM412	3:30 PM	876-54-3210	TTH
9876	IS460	SPRING	2006	BLM307	1:30 PM	654-32-1098	TTH

FIGURE 3.1
Matching Values among the Enrollment, Offering, and Student Tables



example, the first and third rows of the *Enrollment* table have the same *StdSSN* value (123-45-6789) as the first row of the *Student* table. Likewise, the values in the *OfferNo* column of the *Enrollment* table match the *OfferNo* column in the *Offering* table (Table 3.4). Figure 3.1 shows a graphical depiction of the matching values.

The concept of matching values is crucial in relational databases. As you will see, relational databases typically contain many tables. Even a modest-size database can have 10 to 15 tables. Large databases can have hundreds of tables. To extract meaningful information, it is often necessary to combine multiple tables using matching values. By matching on *Student.StdSSN* and *Enrollment.StdSSN,* you could combine the *Student* and *Enrollment* tables.[3] Similarly, by matching on *Enrollment.OfferNo* and *Offering.OfferNo,*

[3] When columns have identical names in two tables, it is customary to precede the column name with the table name and a period as *Student.StdSSN* and *Enrollment.StdSSN*.

TABLE 3.5
Alternative Terminology for Relational Databases

Table-Oriented	Set-Oriented	Record-Oriented
Table	Relation	Record type, file
Row	Tuple	Record
Column	Attribute	Field

you could combine the *Enrollment* and *Offering* tables. As you will see later in this chapter, the operation of combining tables on matching values is known as a join. Understanding the connections between tables (or ways that tables can be combined) is crucial for extracting useful data.

3.1.3 Alternative Terminology

You should be aware that other terminology is used besides table, row, and column. Table 3.5 shows three roughly equivalent terminologies. The divergence in terminology is due to the different groups that use databases. The table-oriented terminology appeals to end users; the set-oriented terminology appeals to academic researchers; and the record-oriented terminology appeals to information systems professionals. In practice, these terms may be mixed. For example, in the same sentence you might hear both "tables" and "fields." You should expect to see a mix of terminology in your career.

3.2 Integrity Rules

In the previous section, you learned that a relational database consists of a collection of interrelated tables. To ensure that a database provides meaningful information, integrity rules are necessary. This section describes two important integrity rules (entity integrity and referential integrity), examples of their usage, and a notation to visualize referential integrity.

3.2.1 Definition of the Integrity Rules

Entity integrity means that each table must have a column or combination of columns with unique values.[4] Unique means that no two rows of a table have the same value. For example, *StdSSN* in *Student* is unique and the combination of *StdSSN* and *OfferNo* is unique in *Enrollment*. Entity integrity ensures that entities (people, things, and events) are uniquely identified in a database. For auditing, security, and communication reasons, it is important that business entities be easily traceable.

Referential integrity means that the column values in one table must match column values in a related table. For example, the value of *StdSSN* in each row of the *Enrollment* table must match the value of *StdSSN* in some row of the *Student* table. Referential integrity ensures that a database contains valid connections. For example, it is critical that each row of the *Enrollment* table contains a Social Security number of a valid student. Otherwise, some enrollments can be meaningless, possibly resulting in students denied enrollment because nonexisting students took their places.

For more precise definitions of entity integrity and referential integrity, some other definitions are necessary. These prerequisite definitions and the more precise definitions follow.

Definitions

- Superkey: a column or combination of columns containing unique values for each row. The combination of every column in a table is always a superkey because rows in a table must be unique.[5]

[4] Entity integrity is also known as uniqueness integrity.

[5] The uniqueness of rows is a feature of the relational model that SQL does not require.

- Candidate key: a minimal superkey. A superkey is minimal if removing any column makes it no longer unique.
- Null value: a special value that represents the absence of an actual value. A null value can mean that the actual value is unknown or does not apply to the given row.
- Primary key: a specially designated candidate key. The primary key for a table cannot contain null values.
- Foreign key: a column or combination of columns in which the values must match those of a candidate key. A foreign key must have the same data type as its associated candidate key. In the CREATE TABLE statement of SQL, a foreign key must be associated with a primary key rather than merely a candidate key.

Integrity Rules

- Entity integrity rule: No two rows of a table can contain the same value for the primary key. In addition, no row can contain a null value for any column of a primary key.
- Referential integrity rule: Only two kinds of values can be stored in a foreign key:
  - a value matching a candidate key value in some row of the table containing the associated candidate key or
  - a null value.

3.2.2 Application of the Integrity Rules

To extend your understanding, let us apply the integrity rules to several tables in the university database. The primary key of *Student* is *StdSSN*. A primary key can be designated as part of the CREATE TABLE statement. To designate *StdSSN* as the primary key of *Student,* you use a CONSTRAINT clause for the primary key at the end of the CREATE TABLE statement. The constraint name (PKStudent) following the CONSTRAINT keyword facilitates identification of the constraint if a violation occurs when a row is inserted or updated.

```
CREATE TABLE Student
(       StdSSN              CHAR(11),
        StdFirstName        VARCHAR(50),
        StdLastName         VARCHAR(50),
        StdCity             VARCHAR(50),
        StdState            CHAR(2),
        StdZip              CHAR(10),
        StdMajor            CHAR(6),
        StdClass            CHAR(2),
        StdGPA              DECIMAL(3,2),
CONSTRAINT PKStudent PRIMARY KEY (StdSSN)    )
```

Social Security numbers are assigned by the federal government so the university does not have to assign them. In other cases, primary values are assigned by an organization. For example, customer numbers, product numbers, and employee numbers are typically assigned by the organization controlling the underlying database. In these cases, automatic generation of unique values is required. Some DBMSs support automatic generation of unique values as explained in Appendix 3.C.

Entity Integrity Variations

Candidate keys that are not primary keys are declared with the UNIQUE keyword. The *Course* table (see Table 3.6) contains two candidate keys: *CourseNo* (primary key) and *CrsDesc* (course description). The *CourseNo* column is the primary key because it is more

TABLE 3.6
Sample Course Table

CourseNo	CrsDesc	CrsUnits
IS320	FUNDAMENTALS OF BUSINESS	4
IS460	SYSTEMS ANALYSIS	4
IS470	BUSINESS DATA COMMUNICATIONS	4
IS480	FUNDAMENTALS OF DATABASE	4

stable than the *CrsDesc* column. Course descriptions may change over time, but course numbers remain the same.

```
CREATE TABLE Course
                (       CourseNo            CHAR(6),
                        CrsDesc             VARCHAR(250),
                        CrsUnits            SMALLINT,
CONSTRAINT PKCourse PRIMARY KEY(CourseNo),
CONSTRAINT UniqueCrsDesc UNIQUE (CrsDesc)  )
```

Some tables need more than one column in the primary key. In the *Enrollment* table, the combination of *StdSSN* and *OfferNo* is the only candidate key. Both columns are needed to identify a row. A primary key consisting of more than one column is known as a composite or a combined primary key.

```
CREATE TABLE Enrollment
                (       OfferNo             INTEGER,
                        StdSSN              CHAR(11),
                        EnrGrade            DECIMAL(3,2),
CONSTRAINT PKEnrollment PRIMARY KEY(OfferNo, StdSSN)    )
```

Nonminimal superkeys are not important because they are common and contain columns that do not contribute to the uniqueness property. For example, the combination of *StdSSN* and *StdLastName* is unique. However, if *StdLastName* is removed, *StdSSN* is still unique.

Referential Integrity

For referential integrity, the columns *StdSSN* and *OfferNo* are foreign keys in the *Enrollment* table. The *StdSSN* column refers to the *Student* table and the *OfferNo* column refers to the *Offering* table (Table 3.4). An *Offering* row represents a course given in an academic period (summer, winter, etc.), year, time, location, and days of the week. The primary key of *Offering* is *OfferNo*. A course such as IS480 will have different offer numbers each time it is offered.

Referential integrity constraints can be defined similarly to the way of defining primary keys. For example, to define the foreign keys in *Enrollment,* you should use CONSTRAINT clauses for foreign keys at the end of the CREATE TABLE statement as shown in the revised CREATE TABLE statement for the *Enrollment* table.

```
CREATE TABLE Enrollment
                (       OfferNo             INTEGER,
                        StdSSN              CHAR(11),
                        EnrGrade            DECIMAL(3,2),
CONSTRAINT PKEnrollment PRIMARY KEY(OfferNo, StdSSN),
CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo) REFERENCES Offering,
CONSTRAINT FKStdSSN FOREIGN KEY (StdSSN) REFERENCES Student    )
```

Although referential integrity permits foreign keys to have null values, it is not common for foreign keys to have null values. When a foreign key is part of a primary key, null values are not permitted because of the entity integrity rule. For example, null values are not permitted for either *Enrollment.StdSSN* or *Enrollment.OfferNo* because each column is part of the primary key.

When a foreign key is not part of a primary key, usage dictates whether null values should be permitted. For example, *Offering.CourseNo,* a foreign key referring to *Course* (Table 3.4), is not part of a primary key yet null values are not permitted. In most universities, a course cannot be offered before it is approved. Thus, an offering should not be inserted without a related course.

The NOT NULL keywords indicate that a column cannot have null values as shown in the CREATE TABLE statement for the *Offering* table. The NOT NULL constraints are inline constraints associated with a specific column. In contrast, the primary and foreign key constraints in the CREATE TABLE statement for the *Offering* table are table constraints in which the associated columns must be specified in the constraint. Constraint names should be used with both table and inline constraints to facilitate identification when a violation occurs.

```
CREATE TABLE Offering
(       OfferNo             INTEGER,
        CourseNo            CHAR(6)         CONSTRAINT OffCourseNoRequired NOT NULL,
        OffLocation         VARCHAR(50),
        OffDays             CHAR(6),
        OffTerm             CHAR(6)         CONSTRAINT OffTermRequired NOT NULL,
        OffYear             INTEGER         CONSTRAINT OffYearRequired NOT NULL,
        FacSSN              CHAR(11),
        OffTime             DATE,
CONSTRAINT PKOffering   PRIMARY KEY (OfferNo),
CONSTRAINT FKCourseNo FOREIGN KEY(CourseNo) REFERENCES Course,
CONSTRAINT FKFacSSN   FOREIGN KEY(FacSSN) REFERENCES Faculty    )
```

In contrast, *Offering.FacSSN* referring to the faculty member teaching the offering may be null. The *Faculty* table (Table 3.7) stores data about instructors of courses. A null value for *Offering.FacSSN* means that a faculty member is not yet assigned to teach the offering. For example, an instructor is not assigned in the first and third rows of Table 3.4. Because offerings must be scheduled perhaps a year in advance, it is likely that instructors for some offerings will not be known until after the offering row is initially stored. Therefore, permitting null values in the *Offering* table is prudent.

self-referencing relationship
a relationship in which a foreign key refers to the same table. Self-referencing relationships represent associations among members of the same set.

Referential Integrity for Self-Referencing (Unary) Relationships

A referential integrity constraint involving a single table is known as a self-referencing relationship or unary relationship. Self-referencing relationships are not common, but they are important when they occur. In the university database, a faculty member can supervise other faculty members and be supervised by a faculty member. For example, Victoria Emmanuel

TABLE 3.7 Sample Faculty Table

FacSSN	FacFirstName	FacLastName	FacCity	FacState	FacDept	FacRank	FacSalary	FacSupervisor	FacHireDate	FacZipCode
098-76-5432	LEONARD	VINCE	SEATTLE	WA	MS	ASST	$35,000	654-32-1098	01-Apr-95	98111-9921
543-21-0987	VICTORIA	EMMANUEL	BOTHELL	WA	MS	PROF	$120,000		01-Apr-96	98011-2242
654-32-1098	LEONARD	FIBON	SEATTLE	WA	MS	ASSC	$70,000	543-21-0987	01-Apr-95	98121-0094
765-43-2109	NICKI	MACON	BELLEVUE	WA	FIN	PROF	$65,000		01-Apr-97	98015-9945
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	MS	ASST	$40,000	654-32-1098	01-Apr-99	98114-1332
987-65-4321	JULIA	MILLS	SEATTLE	WA	FIN	ASSC	$75,000	765-43-2109	01-Apr-00	98114-9954

(second row) supervises Leonard Fibon (third row). The *FacSupervisor* column shows this relationship: the *FacSupervisor* value in the third row (543-21-0987) matches the *FacSSN* value in the second row. A referential integrity constraint involving the *FacSupervisor* column represents the self-referencing relationship. In the CREATE TABLE statement, the referential integrity constraint for a self-referencing relationship can be written the same way as other referential integrity constraints.

```
CREATE TABLE Faculty
(       FacSSN          CHAR(11),
        FacFirstName    VARCHAR(50)   CONSTRAINT FacFirstNameRequired NOT NULL,
        FacLastName     VARCHAR(50)   CONSTRAINT FacLastNameRequired NOT NULL,
        FacCity         VARCHAR(50)   CONSTRAINT FacCityRequired NOT NULL,
        FacState        CHAR(2)       CONSTRAINT FacStateRequired NOT NULL,
        FacZipCode      CHAR(10)      CONSTRAINT FacZipRequired NOT NULL,
        FacHireDate     DATE,
        FacDept         CHAR(6),
        FacRank         CHAR(4),
        FacSalary       DECIMAL(10,2),
        FacSupervisor   CHAR(11),
CONSTRAINT PKFaculty PRIMARY KEY (FacSSN),
CONSTRAINT FKFacSupervisor FOREIGN KEY (FacSupervisor) REFERENCES Faculty      )
```

3.2.3 Graphical Representation of Referential Integrity

In recent years, commercial DBMSs have provided graphical representations for referential integrity constraints. The graphical representation makes referential integrity easier to define and understand than the text representation in the CREATE TABLE statement. In addition, a graphical representation supports nonprocedural data access.

To depict a graphical representation, let us study the Relationship window in Microsoft Access. Access provides the Relationship window to visually define and display referential integrity constraints. Figure 3.2 shows the Relationship window for the tables of the university database. Each line represents a referential integrity constraint or relationship. In a relationship, the primary key table is known as the parent or "1" table (for example,

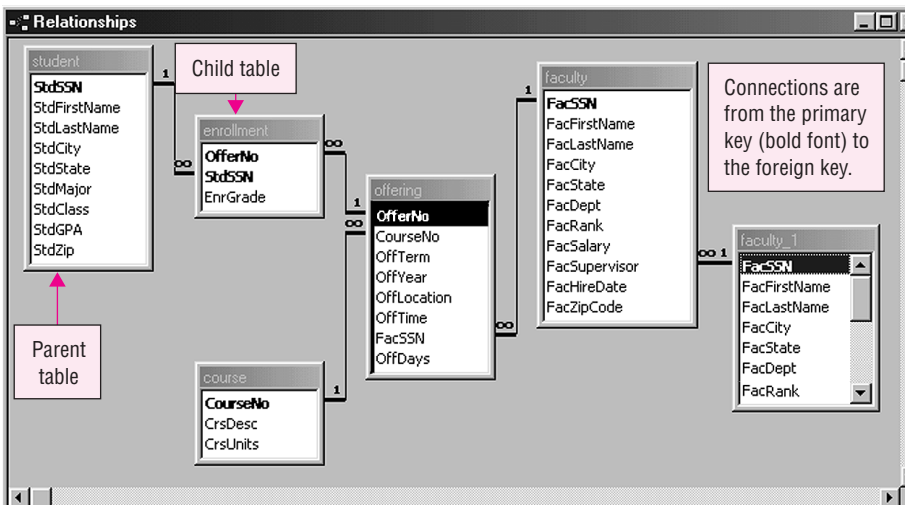


FIGURE 3.2
Relationship Window for the University Database

Student) and the foreign key table (for example, *Enrollment*) is known as the child or "M" (many) table.

1-M relationship
a connection between two tables in which one row of a parent table can be referenced by many rows of a child table. 1-M relationships are the most common kind of relationship.

The relationship from *Student* to *Enrollment* is called "1-M" (one to many) because a student can be related to many enrollments but an enrollment can be related to only one student. Similarly, the relationship from the *Offering* table to the *Enrollment* table means that an offering can be related to many enrollments but an enrollment can be related to only one offering. You should practice by writing similar sentences for the other relationships in Figure 3.2.

M-N relationship
a connection between two tables in which rows of each table can be related to many rows of the other table. M-N relationships cannot be directly represented in the Relational Model. Two 1-M relationships and a linking or associative table represent an M-N relationship.

M-N (many to many) relationships are not directly represented in the Relational Model. An M-N relationship means that rows from each table can be related to many rows of the other table. For example, a student enrolls in many course offerings and a course offering contains many students. In the Relational Model, a pair of 1-M relationships and a linking or associative table represents an M-N relationship. In Figure 3.2, the linking table *Enrollment* and its relationships with *Offering* and *Student* represent an M-N relationship between the *Student* and *Offering* tables.

Self-referencing relationships are represented indirectly in the Relationship window. The self-referencing relationship involving *Faculty* is represented as a relationship between the *Faculty* and *Faculty_1* tables. *Faculty_1* is not a real table as it is created only inside the Access Relationship window. Access can only indirectly show self-referencing relationships.

A graphical representation such as the Relationship window makes it easy to identify tables that should be combined to answer a retrieval request. For example, assume that you want to find instructors who teach courses with "database" in the course description. Clearly, you need the *Course* table to find "database" courses. You also need the *Faculty* table to display instructor data. Figure 3.2 shows that you also need the *Offering* table because *Course* and *Faculty* are not directly connected. Rather, *Course* and *Faculty* are connected through *Offering*. Thus, visualizing relationships helps to identify tables needed to fulfill retrieval requests. Before attempting the retrieval problems in later chapters, you should carefully study a graphical representation of the relationships. You should construct your own diagram if one is not available.

3.3 Delete and Update Actions for Referenced Rows

For each referential integrity constraint, you should carefully consider actions on referenced rows in parent tables of 1-M relationships. A parent row is referenced if there are rows in a child table with foreign key values identical to the primary key value of the parent table row. For example, the first row of the *Course* table (Table 3.6) with *CourseNo* "IS320" is referenced by the first row of the *Offering* table (Table 3.4). It is natural to consider what happens to related *Offering* rows when the referenced *Course* row is deleted or the *CourseNo* is updated. More generally, these concerns can be stated as

Deleting a referenced row: What happens to related rows (that is, rows in the child table with the identical foreign key value) when the referenced row in the parent table is deleted?

Updating the primary key of a referenced row: What happens to related rows when the primary key of the referenced row in the parent table is updated?

Actions on referenced rows are important when changing the rows of a database. When developing data entry forms (discussed in Chapter 10), actions on referenced rows can be especially important. For example, if a data entry form permits deletion of rows in the *Course* table, actions on related rows in the *Offering* table must be carefully planned. Otherwise, the database can become inconsistent or difficult to use.

Possible Actions

There are several possible actions in response to the deletion of a referenced row or the update of the primary key of a referenced row. The appropriate action depends on the tables involved. The following list describes the actions and provides examples of usage.

- **Restrict[6]:** Do not allow the action on the referenced row. For example, do not permit a *Student* row to be deleted if there are any related *Enrollment* rows. Similarly, do not allow *Student.StdSSN* to be changed if there are related *Enrollment* rows.
- **Cascade:** Perform the same action (cascade the action) to related rows. For example, if a *Student* is deleted, then delete the related *Enrollment* rows. Likewise, if *Student.StdSSN* is changed in some row, update *StdSSN* in the related *Enrollment* rows.
- **Nullify:** Set the foreign key of related rows to null. For example, if a *Faculty* row is deleted, then set *FacSSN* to NULL in related *Offering* rows. Likewise, if *Faculty.FacSSN* is updated, then set *FacSSN* to NULL in related *Offering* rows. The nullify action is not permitted if the foreign key does not allow null values. For example, the nullify option is not valid when deleting rows of the *Student* table because *Enrollment.StdSSN* is part of the primary key.
- **Default:** Set the foreign key of related rows to its default value. For example, if a *Faculty* row is deleted, then set *FacSSN* to a default faculty in related *Offering* rows. The default faculty might have an interpretation such as "to be announced." Likewise, if *Faculty.FacSSN* is updated, then set *FacSSN* to its default value in related *Offering* rows. The default action is an alternative to the nullify action as the default action avoids null values.

The delete and update actions can be specified in SQL using the ON DELETE and ON UPDATE clauses. These clauses are part of foreign key constraints. For example, the revised CREATE TABLE statement for the *Enrollment* table shows ON DELETE and ON UPDATE clauses for the *Enrollment* table. The RESTRICT keyword means restrict (the first possible action). The keywords CASCADE, SET NULL, and SET DEFAULT can be used to specify the second through fourth options, respectively.

```
CREATE TABLE Enrollment
(       OfferNo             INTEGER,
        StdSSN              CHAR(11),
        EnrGrade            DECIMAL(3,2),
CONSTRAINT PKEnrollment  PRIMARY KEY (OfferNo, StdSSN),
CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo) REFERENCES Offering
    ON DELETE RESTRICT
    ON UPDATE CASCADE,
CONSTRAINT FKStdSSN FOREIGN KEY (StdSSN) REFERENCES Student
    ON DELETE RESTRICT
    ON UPDATE CASCADE       )
```

Before finishing this section, you should understand the impact of referenced rows on insert operations. A referenced row must be inserted before its related rows. For example, before inserting a row in the *Enrollment* table, the referenced rows in the *Student* and *Offering* tables must exist. Referential integrity places an ordering on insertion of rows from different tables. When designing data entry forms, you should carefully consider the impact of referential integrity on the order that users complete forms.

[6] There is a related action designated by the keywords NO ACTION. The difference between RESTRICT and NO ACTION involves the concept of deferred integrity constraints, discussed in Chapter 15.

3.4 Operators of Relational Algebra

In previous sections of this chapter, you have studied the terminology and integrity rules of relational databases with the goal of understanding existing relational databases. In particular, understanding connections among tables was emphasized as a prerequisite to retrieving useful information. This section describes some fundamental operators that can be used to retrieve useful information from a relational database.

You can think of relational algebra similarly to the algebra of numbers except that the objects are different: algebra applies to numbers and relational algebra applies to tables. In algebra, each operator transforms one or more numbers into another number. Similarly, each operator of relational algebra transforms a table (or two tables) into a new table.

This section emphasizes the study of each relational algebra operator in isolation. For each operator, you should understand its purpose and inputs. While it is possible to combine operators to make complicated formulas, this level of understanding is not important for developing query formulation skills. Using relational algebra by itself to write queries can be awkward because of details such as ordering of operations and parentheses. Therefore, you should seek only to understand the meaning of each operator, not how to combine operators to write expressions.

The coverage of relational algebra groups the operators into three categories. The most widely used operators (restrict, project, and join) are presented first. The extended cross product operator is also presented to provide background for the join operator. Knowledge of these operators will help you to formulate a large percentage of queries. More specialized operators are covered in latter parts of the section. The more specialized operators include the traditional set operators (union, intersection, and difference) and advanced operators (summarize and divide). Knowledge of these operators will help you formulate more difficult queries.

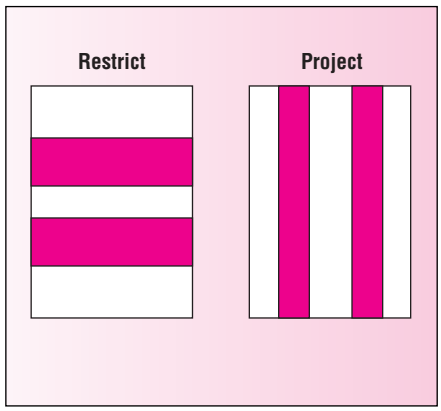
3.4.1 Restrict (Select) and Project Operators

restrict
an operator that retrieves a subset of the rows of the input table that satisfy a given condition.

The restrict (also known as select) and project operators produce subsets of a table.[7] Because users often want to see a subset rather than an entire table, these operators are widely used. These operators are also popular because they are easy to understand.

The restrict and project operators produce an output table that is a subset of an input table (Figure 3.3). Restrict produces a subset of the rows, while project produces a subset of

FIGURE 3.3
Graphical Representation of Restrict and Project Operators



[7] In this book, the operator name restrict is used to avoid confusion with the SQL SELECT statement. The operator is more widely known as select.

TABLE 3.8 **Result of Restrict Operation on the Sample Offering Table (TABLE 3.4)**

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	FacSSN	OffDays
3333	IS320	SPRING	2006	BLM214	8:30 AM	098-76-5432	MW
5678	IS480	SPRING	2006	BLM302	10:30 AM	987-65-4321	MW

TABLE 3.9
Result of a Project Operation on *Offering.CourseNo*

CourseNo
IS320
IS460
IS480

the columns. Restrict uses a condition or logical expression to indicate what rows should be retained in the output. Project uses a list of column names to indicate what columns to retain in the output. Restrict and project are often used together because tables can have many rows and columns. It is rare that a user wants to see all rows and columns.

project
an operator that retrieves a specified subset of the columns of the input table.

The logical expression used in the restrict operator can include comparisons involving columns and constants. Complex logical expressions can be formed using the logical operators AND, OR, and NOT. For example, Table 3.8 shows the result of a restrict operation on Table 3.4 where the logical expression is: OffDays = 'MW' AND OffTerm = 'SPRING' AND OffYear = 2006.

A project operation can have a side effect. Sometimes after a subset of columns is retrieved, there are duplicate rows. When this occurs, the project operator removes the duplicate rows. For example, if *Offering.CourseNo* is the only column used in a project operation, only three rows are in the result (Table 3.9) even though the *Offering* table (Table 3.4) has nine rows. The column *Offering.CourseNo* contains only three unique values in Table 3.4. Note that if the primary key or a candidate key is included in the list of columns, the resulting table has no duplicates. For example, if *OfferNo* was included in the list of columns, the result table would have nine rows with no duplicate removal necessary.

This side effect is due to the mathematical nature of relational algebra. In relational algebra, tables are considered sets. Because sets do not have duplicates, duplicate removal is a possible side effect of the project operator. Commercial languages such as SQL usually take a more pragmatic view. Because duplicate removal can be computationally expensive, duplicates are not removed unless the user specifically requests it.

3.4.2 Extended Cross Product Operator

The extended cross product operator can combine any two tables. Other table combining operators have conditions about the tables to combine. Because of its unrestricted nature, the extended cross product operator can produce tables with excessive data. The extended cross product operator is important because it is a building block for the join operator. When you initially learn the join operator, knowledge of the extended cross product operator can be useful. After you gain experience with the join operator, you will not need to rely on the extended cross product operator.

extended cross product
an operator that builds a table consisting of all combinations of rows from each of the two input tables.

The extended cross product (product for short) operator shows everything possible from two tables.[8] The product of two tables is a new table consisting of all possible combinations of rows from the two input tables. Figure 3.4 depicts a product of two single column tables. Each result row consists of the columns of the *Faculty* table (only *FacSSN*) and the columns of the *Student* table (only *StdSSN*). The name of the operator (product) derives from the

[8] The extended cross product operator is also known as the Cartesian product after French mathematician René Descartes.

number of rows in the result. The number of rows in the resulting table is the product of the number of rows of the two input tables. In contrast, the number of result columns is the sum of the columns of the two input tables. In Figure 3.4, the result table has nine rows and two columns.

As another example, consider the product of the sample *Student* (Table 3.10) and *Enrollment* (Table 3.11) tables. The resulting table (Table 3.12) has 9 rows (3 × 3) and 7 columns (4 + 3). Note that most rows in the result are not meaningful as only three rows have the same value for *StdSSN*.

FIGURE 3.4
Cross Product Example

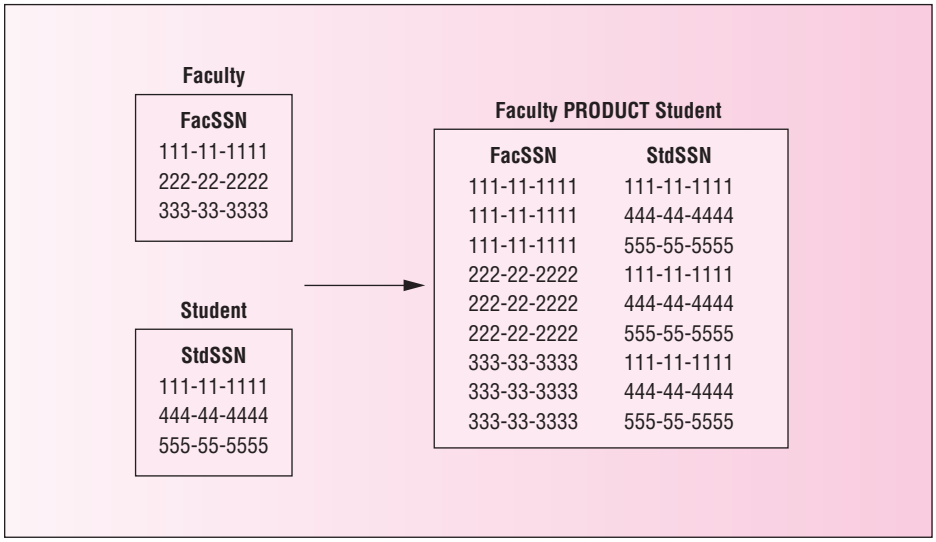


TABLE 3.10
Sample *Student* Table

StdSSN	StdLastName	StdMajor	StdClass
123-45-6789	WELLS	IS	FR
124-56-7890	NORBERT	FIN	JR
234-56-7890	KENDALL	ACCT	JR

TABLE 3.11
Sample *Enrollment* Table

OfferNo	StdSSN	EnrGrade
1234	123-45-6789	3.3
1234	234-56-7890	3.5
4321	124-56-7890	3.2

TABLE 3.12 ***Student* PRODUCT *Enrollment***

Student.StdSSN	StdLastName	StdMajor	StdClass	OfferNo	Enrollment.StdSSN	EnrGrade
123-45-6789	WELLS	IS	FR	1234	123-45-6789	3.3
123-45-6789	WELLS	IS	FR	1234	234-56-7890	3.5
123-45-6789	WELLS	IS	FR	4321	124-56-7890	3.2
124-56-7890	NORBERT	FIN	JR	1234	123-45-6789	3.3
124-56-7890	NORBERT	FIN	JR	1234	234-56-7890	3.5
124-56-7890	NORBERT	FIN	JR	4321	124-56-7890	3.2
234-56-7890	KENDALL	ACCT	JR	1234	123-45-6789	3.3
234-56-7890	KENDALL	ACCT	JR	1234	234-56-7890	3.5
234-56-7890	KENDALL	ACCT	JR	4321	124-56-7890	3.2

As these examples show, the extended cross product operator often generates excessive data. Excessive data are as bad as lack of data. For example, the product of a student table of 30,000 rows and an enrollment table of 300,000 rows is a table of nine billion rows! Most of these rows would be meaningless combinations. So it is rare that a cross product operation by itself is needed. Rather, the importance of the cross product operator is as a building block for other operators such as the join operator.

3.4.3 Join Operator

Join is the most widely used operator for combining tables. Because most databases have many tables, combining tables is important. Join differs from cross product because join requires a matching condition on rows of two tables. Most tables are combined in this way. To a large extent, your skill in retrieving useful data will depend on your ability to use the join operator.

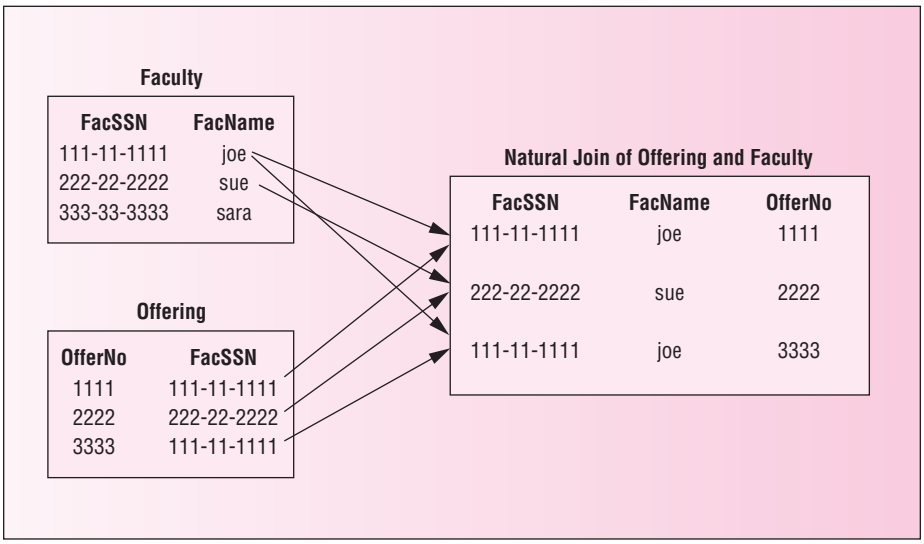
join
an operator that produces a table containing rows that match on a condition involving a column from each input table.

The join operator builds a new table by combining rows from two tables that match on a join condition. Typically, the join condition specifies that two rows have an identical value in one or more columns. When the join condition involves equality, the join is known as an equi-join, for equality join. Figure 3.5 shows a join of sample *Faculty* and *Offering* tables where the join condition is that the *FacSSN* columns are equal. Note that only a few columns are shown to simplify the illustration. The arrows indicate how rows from the input tables combine to form rows in the result table. For example, the first row of the *Faculty* table combines with the first and third rows of the *Offering* table to yield two rows in the result table.

natural join
a commonly used join operator where the matching condition is equality (equi-join), one of the matching columns is discarded in the result table, and the join columns have the same unqualified names.

The natural join operator is the most common join operation. In a natural join operation, the join condition is equality (equi-join), one of the join columns is removed, and the join columns have the same unqualified name.[9] In Figure 3.5, the result table contains only three columns because the natural join removes one of the *FacSSN* columns. The particular column (*Faculty.FacSSN* or *Offering.FacSSN*) removed does not matter.

FIGURE 3.5
Sample Natural Join Operation



[9] An unqualified name is the column name without the table name. The full name of a column includes the table name. Thus, the full names of the join columns in Figure 3.5 are *Faculty.FacSSN* and *Offering.FacSSN*.

TABLE 3.13
Sample *Student* Table

StdSSN	StdLastName	StdMajor	StdClass
123-45-6789	WELLS	IS	FR
124-56-7890	NORBERT	FIN	JR
234-56-7890	KENDALL	ACCT	JR

TABLE 3.14
Sample *Enrollment* Table

OfferNo	StdSSN	EnrGrade
1234	123-45-6789	3.3
1234	234-56-7890	3.5
4321	124-56-7890	3.2

TABLE 3.15
Natural Join of *Student* and *Enrollment*

Student.StdSSN	StdLastName	StdMajor	StdClass	OfferNo	EnrGrade
123-45-6789	WELLS	IS	FR	1234	3.3
124-56-7890	NORBERT	FIN	JR	4321	3.2
234-56-7890	KENDALL	ACCT	JR	1234	3.5

As another example, consider the natural join of *Student* (Table 3.13) and *Enrollment* (Table 3.14) shown in Table 3.15. In each row of the result, *Student.StdSSN* matches *Enrollment.StdSSN*. Only one of the join columns is included in the result. Arbitrarily, *Student.StdSSN* is shown although *Enrollment.StdSSN* could be included without changing the result.

Derivation of the Natural Join

The natural join operator is not primitive because it can be derived from other operators. The natural join operator consists of three steps:

1. A product operation to combine the rows.
2. A restrict operation to remove rows not satisfying the join condition.
3. A project operation to remove one of the join columns.

To depict these steps, the first step to produce the natural join in Table 3.15 is the product result shown in Table 3.12. The second step is to retain only the matching rows (rows 1, 6, and 8 of Table 3.12). A restrict operation is used with Student.StdSSN = Enrollment.StdSSN as the restriction condition. The final step is to eliminate one of the join columns (*Enrollment.StdSSN*). The project operation includes all columns except for *Enrollment.StdSSN*.

Although the join operator is not primitive, it can be conceptualized directly without its primitive operations. When you are initially learning the join operator, it can be helpful to derive the results using the underlying operations. As an exercise, you are encouraged to derive the result in Figure 3.5. After learning the join operator, you should not need to use the underlying operations.

Visual Formulation of Join Operations

As a query formulation aid, many DBMSs provide a visual way to formulate joins. Microsoft Access provides a visual representation of the join operator using the Query Design window. Figure 3.6 depicts a join between *Student* and *Enrollment* on *StdSSN* using the Query Design window. To form this join, you need only to select the tables. Access determines that you should join over the *StdSSN* column. Access assumes that most joins involve a primary key and foreign key combination. If Access chooses the join condition incorrectly, you can choose other join columns.

FIGURE 3.6
Query Design Window Showing a Join between *Student* and *Enrollment*

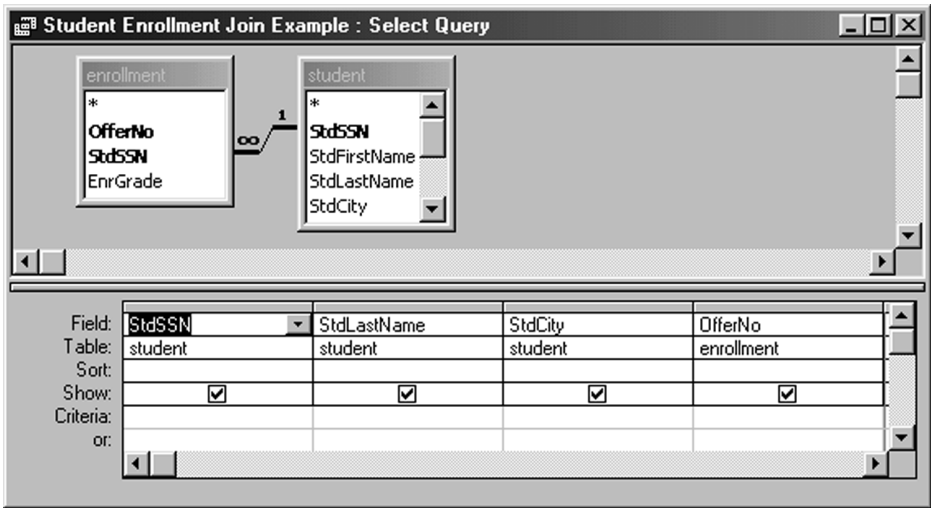
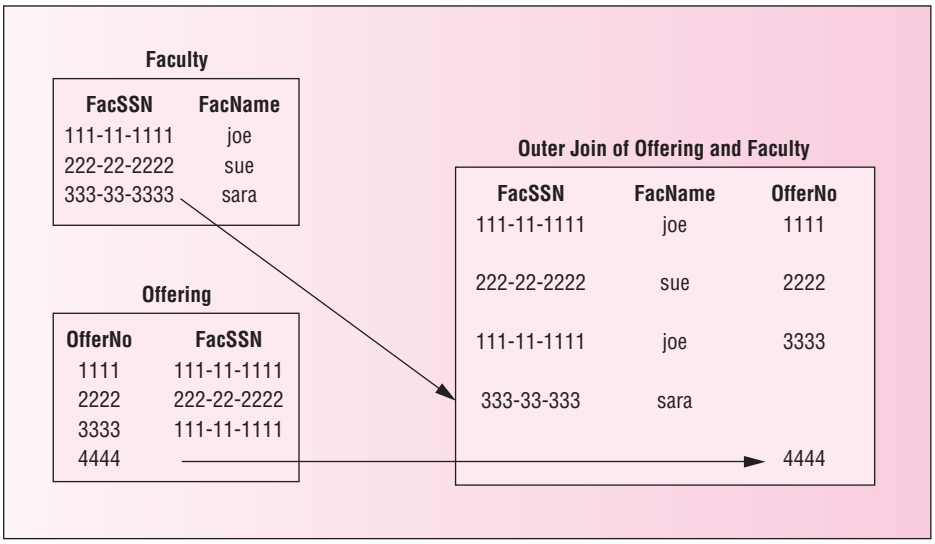


FIGURE 3.7
Sample Outer Join Operation



3.4.4 Outer Join Operator

The result of a join operation includes the rows matching on the join condition. Sometimes it is useful to include both matching and nonmatching rows. For example, you may want to know offerings that have an assigned instructor as well as offerings without an assigned instructor. In these situations, the outer join operator is useful.

The outer join operator provides the ability to preserve nonmatching rows in the result as well as to include the matching rows. Figure 3.7 depicts an outer join between sample *Faculty* and *Offering* tables. Note that each table has one row that does not match any row in the other table. The third row of *Faculty* and the fourth row of *Offering* do not have matching rows in the other table. For nonmatching rows, null values are used to complete the column values in the other table. In Figure 3.7, blanks (no values) represent null values. The fourth result row is the nonmatched row of *Faculty* with a null value for the *OfferNo* column. Likewise, the fifth result row contains a null value for the first two columns because it is a nonmatched row of *Offering*.

Full versus One-Sided Outer Join Operators

full outer join
an operator that produces the matching rows (the join part) as well as the nonmatching rows from both input tables.

one-sided outer join
an operator that produces the matching rows (the join part) as well as the nonmatching rows from the designated input table.

The outer join operator has two variations. The full outer join preserves nonmatching rows from both input tables. Figure 3.7 shows a full outer join because the nonmatching rows from both tables are preserved in the result. Because it is sometimes useful to preserve the nonmatching rows from just one input table, the one-sided outer join operator has been devised. In Figure 3.7, only the first four rows of the result would appear for a one-sided outer join that preserves the rows of the *Faculty* table. The last row would not appear in the result because it is an unmatched row of the *Offering* table. Similarly, only the first three rows and the last row would appear in the result for a one-sided outer join that preserves the rows of the *Offering* table.

The outer join is useful in two situations. A full outer join can be used to combine two tables with some common columns and some unique columns. For example, to combine the *Student* and *Faculty* tables, a full outer join can be used to show all columns about all university people. In Table 3.18, the first two rows are only from the sample *Student* table (Table 3.16), while the last two rows are only from the sample *Faculty* table (Table 3.17). Note the use of null values for the columns from the other table. The third row in Table 3.18 is the row common to the sample *Faculty* and *Student* tables.

A one-sided outer join can be useful when a table has null values in a foreign key. For example, the *Offering* table (Table 3.19) can have null values in the *FacSSN* column representing course offerings without an assigned professor. A one-sided outer join between *Offering* and *Faculty* preserves the rows of *Offering* that do not have an assigned *Faculty* as shown in Table 3.20. With a natural join, the first and third rows of Table 3.20 would not appear. As you will see in Chapter 10, one-sided joins can be useful in data entry forms.

TABLE 3.16
Sample *Student* Table

StdSSN	StdLastName	StdMajor	StdClass
123-45-6789	WELLS	IS	FR
124-56-7890	NORBERT	FIN	JR
876-54-3210	COLAN	MS	SR

TABLE 3.17
Sample *Faculty* Table

FacSSN	FacLastName	FacDept	FacRank
098-76-5432	VINCE	MS	ASST
543-21-0987	EMMANUEL	MS	PROF
876-54-3210	COLAN	MS	ASST

TABLE 3.18 **Result of Full Outer Join of Sample *Student* and *Faculty* Tables**

StdSSN	StdLastName	StdMajor	StdClass	FacSSN	FacLastName	FacDept	FacRank
123-45-6789	WELLS	IS	FR				
124-56-7890	NORBERT	FIN	JR				
876-54-3210	COLAN	MS	SR	876-54-3210	COLAN	MS	ASST
				098-76-5432	VINCE	MS	ASST
				543-21-0987	EMMANUEL	MS	PROF

TABLE 3.19
Sample *Offering* Table

OfferNo	CourseNo	OffTerm	FacSSN
1111	IS320	SUMMER	
1234	IS320	FALL	098-76-5432
2222	IS460	SUMMER	
3333	IS320	SPRING	098-76-5432
4444	IS320	SPRING	543-21-0987

TABLE 3.20 **Result of a One-Sided Outer Join between *Offering* (Table 3.19) and *Faculty* (Table 3.17)**

OfferNo	CourseNo	OffTerm	Offering.FacSSN	Faculty.FacSSN	FacLastName	FacDept	FacRank
1111	IS320	SUMMER					
1234	IS320	FALL	098-76-5432	098-76-5432	VINCE	MS	ASST
2222	IS460	SUMMER					
3333	IS320	SPRING	098-76-5432	098-76-5432	VINCE	MS	ASST
4444	IS320	SPRING	543-21-0987	543-21-0987	EMMANUEL	MS	PROF

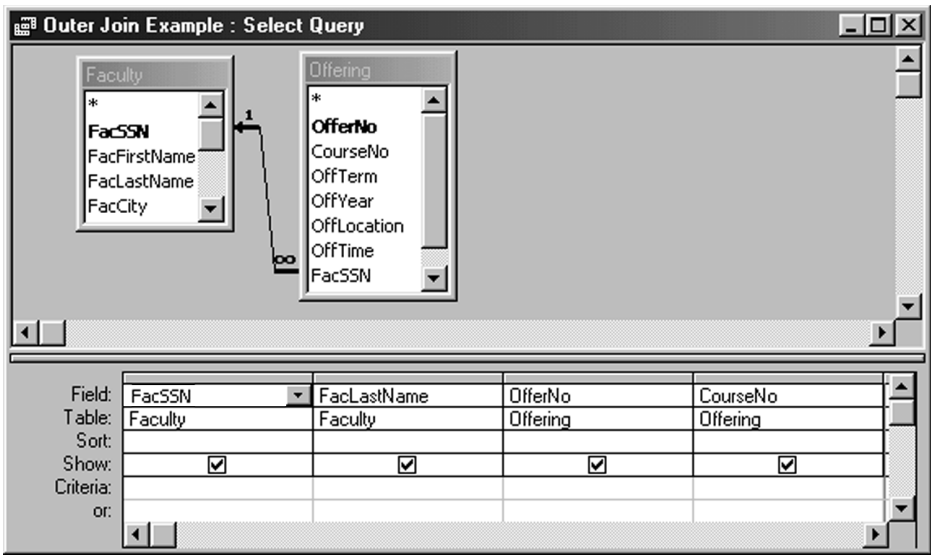


FIGURE 3.8
Query Design Window Showing a One-Sided Outer Join Preserving the *Offering* Table

Visual Formulation of Outer Join Operations

As a query formulation aid, many DBMSs provide a visual way to formulate outer joins. Microsoft Access provides a visual representation of the one-sided join operator in the Query Design window. Figure 3.8 depicts a one-sided outer join that preserves the rows of the *Offering*. The arrow from *Offering* to *Faculty* means that the nonmatched rows of *Offering* are preserved in the result. When combining the *Faculty* and *Offering* tables, Microsoft Access provides three choices: (1) show only the matched rows (a join); (2) show matched rows and nonmatched rows of *Faculty;* and (3) show matched rows and nonmatched rows of *Offering*. Choice (3) is shown in Figure 3.8. Choice (1) would appear similar to Figure 3.6. Choice (2) would have the arrow from *Faculty* to *Offering*.

traditional set operators
the union operator produces a table containing rows from either input table. The intersection operator produces a table containing rows common to both input tables. The difference operator produces a table containing rows from the first input table but not in the second input table.

3.4.5 Union, Intersection, and Difference Operators

The union, intersection, and difference table operators are similar to the traditional set operators. The traditional set operators are used to determine all members of two sets (union), common members of two sets (intersection), and members unique to only one set (difference), as depicted in Figure 3.9.

The union, intersection, and difference operators for tables apply to rows of a table but otherwise operate in the same way as the traditional set operators. A union operation retrieves all the rows in either table. For example, a union operator applied to two student tables at different universities can find all student rows. An intersection operation retrieves just the common rows. For example, an intersection operation can determine the students attending both universities. A difference operation retrieves the rows in the first table but

FIGURE 3.9 **Venn Diagrams for Traditional Set Operators**

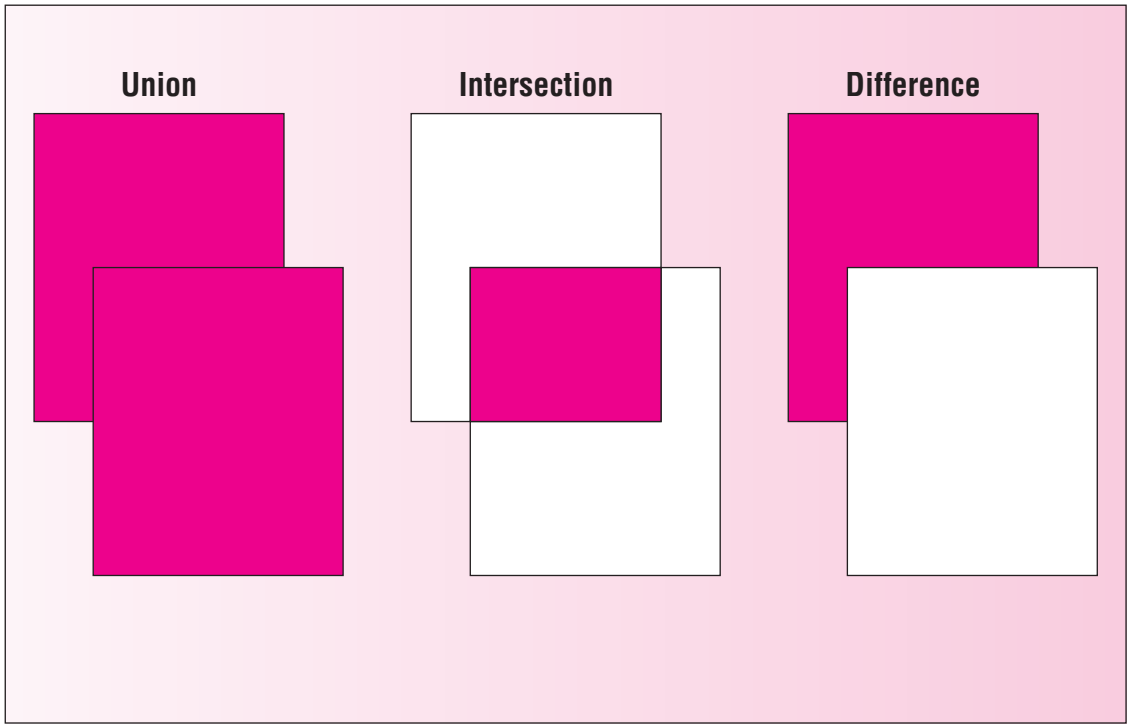


TABLE 3.21
***Student1* Table**

StdSSN	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	SEATTLE	WA	IS	FR	3.00
124-56-7890	NORBERT	BOTHELL	WA	FIN	JR	2.70
234-56-7890	KENDALL	TACOMA	WA	ACCT	JR	3.50

TABLE 3.22
***Student2* Table**

StdSSN	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	SEATTLE	WA	IS	FR	3.00
995-56-3490	BAGGINS	AUSTIN	TX	FIN	JR	2.90
111-56-4490	WILLIAMS	SEATTLE	WA	ACCT	JR	3.40

not in the second table. For example, a difference operation can determine the students attending only one university.

Union Compatibility

union compatibility
a requirement on the input tables for the traditional set operators. Each table must have the same number of columns and each corresponding column must have a compatible data type.

Compatibility is a new concept for the table operators as compared to the traditional set operators. With the table operators, both tables must be union compatible because all columns are compared. Union compatibility means that each table must have the same number of columns and each corresponding column must have a compatible data type. Union compatibility can be confusing because it involves positional correspondence of the columns. That is, the first columns of the two tables must have compatible data types, the second columns must have compatible data type, and so on.

To depict the union, intersection, and difference operators, let us apply them to the *Student1* and *Student2* tables (Tables 3.21 and 3.22). These tables are union compatible because they have identical columns listed in the same order. The results of union,

TABLE 3.23
Student1* UNION *Student2

StdSSN	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	SEATTLE	WA	IS	FR	3.00
124-56-7890	NORBERT	BOTHELL	WA	FIN	JR	2.70
234-56-7890	KENDALL	TACOMA	WA	ACCT	JR	3.50
995-56-3490	BAGGINS	AUSTIN	TX	FIN	JR	2.90
111-56-4490	WILLIAMS	SEATTLE	WA	ACCT	JR	3.40

TABLE 3.24
Student1* INTERSECT *Student2

StdSSN	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
123-45-6789	WELLS	SEATTLE	WA	IS	FR	3.00

TABLE 3.25
Student1* DIFFERENCE *Student2

StdSSN	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA
124-56-7890	NORBERT	BOTHELL	WA	FIN	JR	2.70
234-56-7890	KENDALL	TACOMA	WA	ACCT	JR	3.50

intersection, and difference operators are shown in Tables 3.23 through 3.25, respectively. Even though we can determine that two rows are identical from looking only at *StdSSN,* all columns are compared due to the way that the operators are designed.

Note that the result of *Student1* DIFFERENCE *Student2* would not be the same as *Student2* DIFFERENCE *Student1*. The result of the latter (*Student2* DIFFERENCE *Student1*) is the second and third rows of *Student2* (rows in *Student2* but not in *Student1*).

Because of the union compatibility requirement, the union, intersection, and difference operators are not as widely used as other operators. However, these operators have some important, specialized uses. One use is to combine tables distributed over multiple locations. For example, suppose there is a student table at Big State University (*BSUStudent*) and a student table at University of Big State (*UBSStudent*). Because these tables have identical columns, the traditional set operators are applicable. To find students attending either university, you should use *UBSStudent* UNION *BSUStudent*. To find students only attending Big State, you should use *BSUStudent* DIFFERENCE *UBSStudent*. To find students attending both universities, you should use *UBSStudent* INTERSECT *BSUStudent*. Note that the resulting table in each operation has the same number of columns as the two input tables.

The traditional operators are also useful if there are tables that are similar but not union compatible. For example, the *Student* and *Faculty* tables have some compatible columns (*StdSSN* with *FacSSN, StdLastName* with *FacLastName,* and *StdCity* with *FacCity*), but other columns are different. The union compatible operators can be used if the *Student* and *Faculty* tables are first made union compatible using the project operator presented in Section 3.4.1.

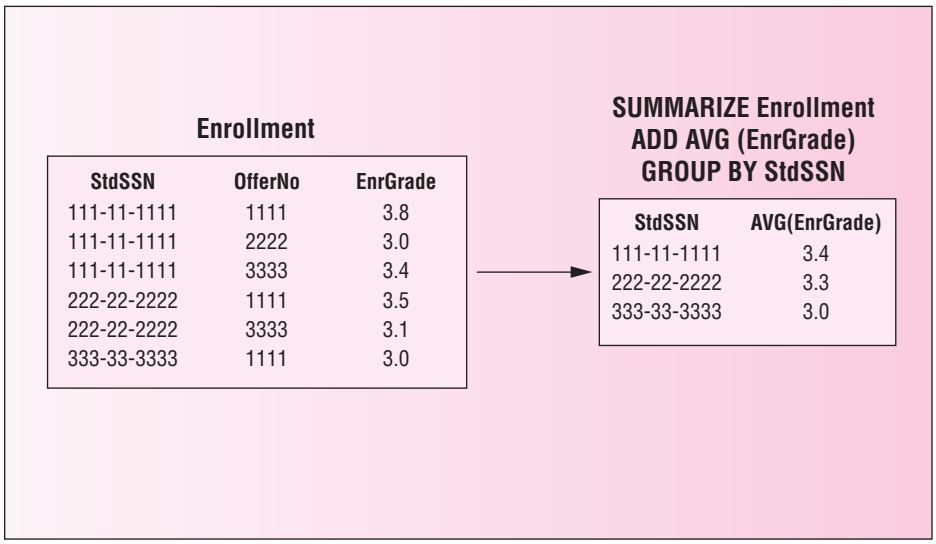
3.4.6 Summarize Operator

summarize
an operator that produces a table with rows that summarize the rows of the input table. Aggregate functions are used to summarize the rows of the input table.

Summarize is a powerful operator for decision making. Because tables can contain many rows, it is often useful to see statistics about groups of rows rather than individual rows. The summarize operator allows groups of rows to be compressed or summarized by a calculated value. Almost any kind of statistical function can be used to summarize groups of rows. Because this is not a statistics book, we will use only simple functions such as count, min, max, average, and sum.

The summarize operator compresses a table by replacing groups of rows with individual rows containing calculated values. A statistical or aggregate function is used for the

FIGURE 3.10
Sample Summarize Operation



divide
an operator that produces a table in which the values of a column from one input table are associated with all the values from a column of a second input table.

TABLE 3.26 **Sample *Faculty* Table**

FacSSN	FacLastName	FacDept	FacRank	FacSalary	FacSupervisor	FacHireDate
098-76-5432	VINCE	MS	ASST	$35,000	654-32-1098	01-Apr-95
543-21-0987	EMMANUEL	MS	PROF	$120,000		01-Apr-96
654-32-1098	FIBON	MS	ASSC	$70,000	543-21-0987	01-Apr-94
765-43-2109	MACON	FIN	PROF	$65,000		01-Apr-97
876-54-3210	COLAN	MS	ASST	$40,000	654-32-1098	01-Apr-99
987-65-4321	MILLS	FIN	ASSC	$75,000	765-43-2109	01-Apr-00

TABLE 3.27
Result Table for SUMMARIZE *Faculty* ADD AVG(*FacSalary*) GROUP BY *FacDept*

FacDept	FacSalary
MS	$66,250
FIN	$70,000

calculated values. Figure 3.10 depicts a summarize operation for a sample enrollment table. The input table is grouped on the *StdSSN* column. Each group of rows is replaced by the average of the grade column.

As another example, Table 3.27 shows the result of a summarize operation on the sample *Faculty* table in Table 3.26. Note that the result contains one row per value of the grouping column, *FacDept*.

The summarize operator can include additional calculated values (also showing the minimum salary, for example) and additional grouping columns (also grouping on *FacRank,* for example). When grouping on multiple columns, each result row shows one combination of values for the grouping columns.

3.4.7 Divide Operator

The divide operator is a more specialized and difficult operator than join because the matching requirement in divide is more stringent than join. For example, a join operator is used to retrieve offerings taken by *any* student. A divide operator is required to retrieve

offerings taken by *all* (or every) students. Because divide has more stringent matching conditions, it is not as widely used as join, and it is more difficult to understand. When appropriate, the divide operator provides a powerful way to combine tables.

The divide operator for tables is somewhat analogous to the divide operator for numbers. In numerical division, the objective is to find how many times one number contains another number. In table division, the objective is to find values of one column that contain *every* value in another column. Stated another way, the divide operator finds values of one column that are associated with *every* value in another column.

To understand more concretely how the divide operator works, consider an example with sample *Part* and *SuppPart* (supplier-part) tables as depicted in Figure 3.11. The divide operator uses two input tables. The first table (*SuppPart*) has two columns (a binary table) and the second table (*Part*) has one column (a unary table).[10] The result table has one column where the values come from the first column of the binary table. The result table in Figure 3.11 shows the suppliers who supply every part. The value s3 appears in the output because it is associated with *every* value in the *Part* table. Stated another way, the set of values associated with s3 contains the set of values in the *Part* table.

To understand the divide operator in another way, rewrite the *SuppPart* table as three rows using the angle brackets < > to surround a row: <s3, {p1, p2, p3}>, <s0, {p1}>, <s1, {p2}>. Rewrite the *Part* table as a set: {p1, p2, p3}. The value s3 is in the result table because its set of second column values {p1, p2, p3} contains the values in the second table {p1, p2, p3}. The other *SuppNo* values (s0 and s1) are not in the result because they are not associated with all values in the *Part* table.

As an example using the university database tables, Table 3.30 shows the result of a divide operation involving the sample *Enrollment* (Table 3.28) and *Student* tables (Table 3.29). The result shows offerings in which every student is enrolled. Only *OfferNo* 4235 has all three students enrolled.

FIGURE 3.11
Sample Divide Operation

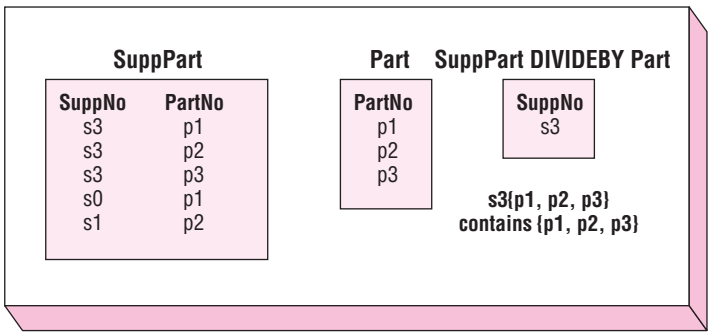


TABLE 3.28
Sample *Enrollment* Table

OfferNo	StdSSN
1234	123-45-6789
1234	234-56-7890
4235	123-45-6789
4235	234-56-7890
4235	124-56-7890
6321	124-56-7890

TABLE 3.29
Sample *Student* Table

StdSSN
123-45-6789
124-56-7890
234-56-7890

TABLE 3.30
Result of *Enrollment* DIVIDEBY *Student*

OfferNo
4235

[10] The divide by operator can be generalized to work with input tables containing more columns. However, the details are not important in this book.

TABLE 3.31
Summary of Meanings of the Relational Algebra Operators

Operator	Meaning
Restrict (Select)	Extracts rows that satisfy a specified condition.
Project	Extracts specified columns.
Product	Builds a table from two tables consisting of all possible combinations of rows, one from each of the two tables.
Union	Builds a table consisting of all rows appearing in either of two tables.
Intersect	Builds a table consisting of all rows appearing in both of two specified tables.
Difference	Builds a table consisting of all rows appearing in the first table but not in the second table.
Join	Extracts rows from a product of two tables such that two input rows contributing to any output row satisfy some specified condition.
Outer Join	Extracts the matching rows (the join part) of two tables and the unmatched rows from both tables.
Divide	Builds a table consisting of all values of one column of a binary (two-column) table that match (in the other column) all values in a unary (one-column) table.
Summarize	Organizes a table on specified grouping columns. Specified aggregate computations are made on each value of the grouping columns.

TABLE 3.32
Summary of Usage of the Relational Algebra Operators

Operator	Notes
Union	Input tables must be union compatible.
Difference	Input tables must be union compatible.
Intersection	Input tables must be union compatible.
Product	Conceptually underlies join operator.
Restrict (Select)	Uses a logical expression.
Project	Eliminates duplicate rows if necessary.
Join	Only matched rows are in the result. Natural join eliminates one join column.
Outer Join	Retains both matched and unmatched rows in the result. Uses null values for some columns of the unmatched rows.
Divide	Stronger operator than join, but less frequently used.
Summarize	Specify grouping column(s) if any and aggregate function(s).

3.4.8 Summary of Operators

To help you recall the relational algebra operators, Tables 3.31 and 3.32 provide a convenient summary of the meaning and usage of each operator. You might want to refer to these tables when studying query formulation in later chapters.

Closing Thoughts

Chapter 3 has introduced the Relational Data Model as a prelude to developing queries, forms, and reports with relational databases. As a first step to work with relational databases, you should understand the basic terminology and integrity rules. You should be able to read table definitions in SQL and other proprietary formats. To effectively query a relational database, you must understand the connections among tables. Most queries involve multiple tables using relationships defined by referential integrity constraints. A graphical representation such as the Relationship window in Microsoft Access provides a powerful

tool to conceptualize referential integrity constraints. When developing applications that can change a database, it is important to respect the action rules for referenced rows.

The final part of this chapter described the operators of relational algebra. At this point, you should understand the purpose of each operator, the number of input tables, and other inputs used. You do not need to write complicated formulas that combine operators. Eventually, you should be comfortable understanding statements such as "write an SQL SELECT statement to join three tables." The SELECT statement will be discussed in Chapters 4 and 9, but the basic idea of a join is important to learn now. As you learn to extract data using the SQL SELECT statement in Chapter 4, you may want to review this chapter again. To help you remember the major points about the operators, the last section of this chapter presented several convenient summaries.

Understanding the operators will improve your knowledge of SQL and your query formulation skills. The meaning of SQL queries can be understood as relational algebra operations. Chapter 4 provides a flowchart demonstrating this correspondence. For this reason, relational algebra provides a yardstick to measure commercial languages: the commercial languages should provide at least the same retrieval ability as the operators of relational algebra.

Review Concepts

- Tables: heading and body.
- Primary keys and entity integrity rule.
- Foreign keys, referential integrity rule, and matching values.
- Visualizing referential integrity constraints.
- Relational Model representation of 1-M relationships, M-N relationships, and self-referencing relationships.
- Actions on referenced rows: cascade, nullify, restrict, default.
- Subset operators: restrict (select) and project.
- Join operator for combining two tables using a matching condition to compare join columns.
- Natural join using equality for the matching operator, join columns with the same unqualified name, and elimination of one join column.
- Most widely used operator for combining tables: natural join.
- Less widely used operators for combining tables: full outer join, one-sided outer join, divide.
- Outer join operator extending the join operator by preserving nonmatching rows.
- One-sided outer join preserving the nonmatching rows of one input table.
- Full outer join preserving the nonmatching rows of both input tables.
- Traditional set operators: union, intersection, difference, extended cross product.
- Union compatibility for comparing rows for the union, intersection, and difference operators.
- Complex matching operator: divide operator for matching on a subset of rows.
- Summarize operator that replaces groups of rows with summary rows.

Questions

1. How is creating a table similar to writing a chapter of a book?
2. With what terminology for relational databases are you most comfortable? Why?
3. What is the difference between a primary key and a candidate key?
4. What is the difference between a candidate key and a superkey?

5. What is a null value?
6. What is the motivation for the entity integrity rule?
7. What is the motivation for the referential integrity rule?
8. What is the relationship between the referential integrity rule and foreign keys?
9. How are candidate keys that are not primary keys indicated in the CREATE TABLE statement?
10. What is the advantage of using constraint names when defining primary key, candidate key, or referential integrity constraints in CREATE TABLE statements?
11. When is it not permissible for foreign keys to store null values?
12. What is the purpose of a database diagram such as the Access Relationship window?
13. How is a 1-M relationship represented in the Relational Model?
14. How is an M-N relationship represented in the Relational Model?
15. What is a self-referencing relationship?
16. How is a self-referencing relationship represented in the Relational Model?
17. What is a referenced row?
18. What two actions on referenced rows can affect related rows in a child table?
19. What are the possible actions on related rows after a referenced row is deleted or its primary key is updated?
20. Why is the restrict action for referenced rows more common than the cascade action?
21. When is the nullify action not allowed?
22. Why study the operators of relational algebra?
23. Why are the restrict and the project operators widely used?
24. Explain how the union, intersection, and difference operators for tables differ from the traditional operators for sets.
25. Why is the join operator so important for retrieving useful information?
26. What is the relationship between the join and the extended cross product operators?
27. Why is the extended cross product operator used sparingly?
28. What happens to unmatched rows with the join operator?
29. What happens to unmatched rows with the full outer join operator?
30. What is the difference between the full outer join and the one-sided outer join?
31. Define a decision-making situation that might require the summarize operator.
32. What is an aggregate function?
33. How are grouping columns used in the summarize operator?
34. Why is the divide operator not as widely used as the join operator?
35. What are the requirements of union compatibility?
36. What are the requirements of the natural join operator?
37. Why is the natural join operator widely used for combining tables?
38. How do visual tools such as the Microsoft Access Query Design tool facilitate the formulation of join operations?

Problems

The problems use the *Customer, OrderTbl,* and *Employee* tables of the simplified Order Entry database. Chapters 4 and 10 extend the database to increase its usefulness. The *Customer* table records clients who have placed orders. The *OrderTbl* contains the basic facts about customer orders. The *Employee* table contains facts about employees who take orders. The primary keys of the tables are *CustNo* for *Customer, EmpNo* for *Employee,* and *OrdNo* for *OrderTbl*.

Customer

CustNo	CustFirstName	CustLastName	CustCity	CustState	CustZip	CustBal
C0954327	Sheri	Gordon	Littleton	CO	80129-5543	$230.00
C1010398	Jim	Glussman	Denver	CO	80111-0033	$200.00
C2388597	Beth	Taylor	Seattle	WA	98103-1121	$500.00
C3340959	Betty	Wise	Seattle	WA	98178-3311	$200.00
C3499503	Bob	Mann	Monroe	WA	98013-1095	$0.00
C8543321	Ron	Thompson	Renton	WA	98666-1289	$85.00

Employee

EmpNo	EmpFirstName	EmpLastName	EmpPhone	EmpEmail
E1329594	Landi	Santos	(303) 789-1234	LSantos@bigco.com
E8544399	Joe	Jenkins	(303) 221-9875	JJenkins@ bigco.com
E8843211	Amy	Tang	(303) 556-4321	ATang@ bigco.com
E9345771	Colin	White	(303) 221-4453	CWhite@ bigco.com
E9884325	Thomas	Johnson	(303) 556-9987	TJohnson@ bigco.com
E9954302	Mary	Hill	(303) 556-9871	MHill@ bigco.com

OrderTbl

OrdNo	OrdDate	CustNo	EmpNo
O1116324	01/23/2007	C0954327	E8544399
O2334661	01/14/2007	C0954327	E1329594
O3331222	01/13/2007	C1010398	
O2233457	01/12/2007	C2388597	E9884325
O4714645	01/11/2007	C2388597	E1329594
O5511365	01/22/2007	C3340959	E9884325
O7989497	01/16/2007	C3499503	E9345771
O1656777	02/11/2007	C8543321	
O7959898	02/19/2007	C8543321	E8544399

1. Write a CREATE TABLE statement for the *Customer* table. Choose data types appropriate for the DBMS used in your course. Note that the *CustBal* column contains numeric data. The currency symbols are not stored in the database. The *CustFirstName* and *CustLastName* columns are required (not null).
2. Write a CREATE TABLE statement for the *Employee* table. Choose data types appropriate for the DBMS used in your course. The *EmpFirstName, EmpLastName,* and *EmpEMail* columns are required (not null).
3. Write a CREATE TABLE statement for the *OrderTbl* table. Choose data types appropriate for the DBMS used in your course. The *OrdDate* column is required (not null).
4. Identify the foreign keys and draw a relationship diagram for the simplified Order Entry database. The *CustNo* column references the *Customer* table and the *EmpNo* column references the *Employee* table. For each relationship, identify the parent table and the child table.
5. Extend your CREATE TABLE statement from problem 3 with referential integrity constraints. Updates and deletes on related rows are restricted.
6. From examination of the sample data and your common understanding of order entry businesses, are null values allowed for the foreign keys in the *OrderTbl* table? Why or why not? Extend the CREATE TABLE statement in problem 5 to enforce the null value restrictions if any.

7. Extend your CREATE TABLE statement for the *Employee* table (problem 2) with a unique constraint for *EmpEMail*. Use a named constraint clause for the unique constraint.
8. Show the result of a restrict operation that lists the orders in February 2007.
9. Show the result of a restrict operation that lists the customers residing in Seattle, WA.
10. Show the result of a project operation that lists the *CustNo, CustFirstName,* and *CustLastName* columns of the *Customer* table.
11. Show the result of a project operation that lists the *CustCity* and *CustState* columns of the *Customer* table.
12. Show the result of a natural join that combines the *Customer* and *OrderTbl* tables.
13. Show the steps to derive the natural join for problem 10. How many rows and columns are in the extended cross product step?
14. Show the result of a natural join of the *Employee* and *OrderTbl* tables.
15. Show the result of a one-sided outer join between the *Employee* and *OrderTbl* tables. Preserve the rows of the *OrderTbl* table in the result.
16. Show the result of a full outer join between the *Employee* and *OrderTbl* tables.
17. Show the result of the restrict operation on *Customer* where the condition is *CustCity* equals "Denver" or "Seattle" followed by a project operation to retain the *CustNo, CustFirstName, CustLastName,* and *CustCity* columns.
18. Show the result of a natural join that combines the *Customer* and *OrderTbl* tables followed by a restrict operation to retain only the Colorado customers (CustState = "CO").
19. Show the result of a summarize operation on *Customer*. The grouping column is *CustState* and the aggregate calculation is COUNT. COUNT shows the number of rows with the same value for the grouping column.
20. Show the result of a summarize operation on *Customer*. The grouping column is *CustState* and the aggregate calculations are the minimum and maximum *CustBal* values.
21. What tables are required to show the *CustLastName, EmpLastName,* and *OrdNo* columns in the result table?
22. Extend your relationship diagram from problem 4 by adding two tables (*OrdLine* and *Product*). Partial CREATE TABLE statements for the primary keys and referential integrity constraints are shown below:

```
CREATE TABLE Product    . . .   PRIMARY KEY (ProdNo)
CREATE TABLE OrdLine    . . .   PRIMARY KEY (OrdNo, ProdNo)
    FOREIGN KEY (OrdNo) REFERENCES Order
    FOREIGN KEY (ProdNo) REFERENCES Product
```

23. Extend your relationship diagram from problem 22 by adding a foreign key in the *Employee* table. The foreign key *SupEmpNo* is the employee number of the supervising employee. Thus, the *SupEmpNo* column references the *Employee* table.
24. What relational algebra operator do you use to find products contained in *every* order? What relational algebra operator do you use to find products contained in *any* order?
25. Are the *Customer* and *Employee* tables union compatible? Why or why not?
26. Using the database after problem 23, what tables must be combined to list the product names on order number O1116324?
27. Using the database after problem 23, what tables must be combined to list the product names ordered by customer number C0954327?
28. Using the database after problem 23, what tables must be combined to list the product names ordered by the customer named Sheri Gordon?
29. Using the database after problem 23, what tables must be combined to list the number of orders submitted by customers residing in Colorado?
30. Using the database after problem 23, what tables must be combined to list the product names appearing on an order taken by an employee named Landi Santos?

References for Further Study

Codd defined the Relational Model in a seminal paper in 1970. His paper inspired research projects at the IBM research laboratories and the University of California at Berkeley that led to commercial relational DBMSs. Date (2003) provides a syntax for the relational algebra. Elmasri and Navathe (2004) provide a more theoretical treatment of the Relational Model, especially the relational algebra.

Appendix 3.A

CREATE TABLE Statements for the University Database Tables

The following are the CREATE TABLE statements for the university database tables (Tables 3.1, 3.3, 3.4, 3.6, and 3.7). The names of the standard data types can vary by DBMS. For example, Microsoft Access SQL supports the TEXT data type instead of CHAR and VARCHAR. In Oracle, you should use VARCHAR2 instead of VARCHAR.

```
CREATE TABLE Student
(    StdSSN              CHAR(11),
     StdFirstName        VARCHAR(50)   CONSTRAINT StdFirstNameRequired NOT NULL,
     StdLastName         VARCHAR(50)   CONSTRAINT StdLastNameRequired NOT NULL,
     StdCity             VARCHAR(50)   CONSTRAINT StdCityRequired NOT NULL,
     StdState            CHAR(2)       CONSTRAINT StdStateRequired NOT NULL,
     StdZip              CHAR(10)      CONSTRAINT StdZipRequired NOT NULL,
     StdMajor            CHAR(6),
     StdClass            CHAR(2),
     StdGPA              DECIMAL(3,2),
CONSTRAINT PKStudent PRIMARY KEY (StdSSN)                    )

CREATE TABLE Course
(    CourseNo            CHAR(6),
     CrsDesc             VARCHAR(250)  CONSTRAINT CrsDescRequired NOT NULL,
     CrsUnits            INTEGER,
CONSTRAINT PKCourse PRIMARY KEY (CourseNo),
CONSTRAINT UniqueCrsDesc UNIQUE (CrsDesc)    )

CREATE TABLE Faculty
(    FacSSN              CHAR(11),
     FacFirstName        VARCHAR(50)   CONSTRAINT FacFirstNameRequired NOT NULL,
     FacLastName         VARCHAR(50)   CONSTRAINT FacLastNameRequired NOT NULL,
     FacCity             VARCHAR(50)   CONSTRAINT FacCityRequired NOT NULL,
     FacState            CHAR(2)       CONSTRAINT FacStateRequired NOT NULL,
     FacZipCode          CHAR(10)      CONSTRAINT FacZipRequired NOT NULL,
     FacHireDate         DATE,
     FacDept             CHAR(6),
     FacRank             CHAR(4),
```

```
    FacSalary           DECIMAL(10,2),
   FacSupervisor        CHAR(11),
CONSTRAINT PKFaculty PRIMARY KEY (FacSSN),
CONSTRAINT FKFacSupervisor FOREIGN KEY (FacSupervisor) REFERENCES Faculty
    ON DELETE SET NULL
    ON UPDATE CASCADE )

CREATE TABLE Offering
(   OfferNo             INTEGER,
    CourseNo            CHAR(6)         CONSTRAINT OffCourseNoRequired NOT NULL,
    OffLocation         VARCHAR(50),
    OffDays             CHAR(6),
    OffTerm             CHAR(6)         CONSTRAINT OffTermRequired NOT NULL,
    OffYear             INTEGER         CONSTRAINT OffYearRequired NOT NULL,
    FacSSN              CHAR(11),
    OffTime             DATE,
CONSTRAINT  PKOffering  PRIMARY KEY (OfferNo),
CONSTRAINT FKCourseNo FOREIGN KEY (CourseNo) REFERENCES Course
    ON DELETE RESTRICT
    ON UPDATE RESTRICT,
CONSTRAINT FKFacSSN FOREIGN KEY (FacSSN) REFERENCES Faculty
    ON DELETE SET NULL
    ON UPDATE CASCADE      )

CREATE TABLE Enrollment
(   OfferNo             INTEGER,
    StdSSN              CHAR(11),
    EnrGrade            DECIMAL(3,2),
CONSTRAINT PKEnrollment PRIMARY KEY (OfferNo, StdSSN),
CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo) REFERENCES Offering
    ON DELETE CASCADE
    ON UPDATE CASCADE,
CONSTRAINT FKStdSSN FOREIGN KEY (StdSSN) REFERENCES Student
    ON DELETE CASCADE
    ON UPDATE CASCADE      )
```

Appendix **3.B**

SQL:2003 Syntax Summary

This appendix provides a convenient summary of the SQL:2003 syntax for the CREATE TABLE statement along with several related statements. For brevity, only the syntax of the most common parts of the statements is described. SQL:2003 is the current version of the SQL standard. The syntax in SQL:2003 for the statements described in this appendix is identical to the syntax in the previous SQL standards, SQL:1999 and SQL-92. For the complete syntax, refer to a SQL:2003 or a SQL-92 reference book such as Groff and

Weinberg (2002). The conventions used in the syntax notation are listed before the statement syntax:

- Uppercase words denote reserved words.
- Mixed-case words without hyphens denote names that the user substitutes.
- The asterisk* after a syntax element indicates that a comma-separated list can be used.
- The plus symbol + after a syntax element indicates that a list can be used. No commas appear in the list.
- Names enclosed in angle brackets < > denote definitions defined later in the syntax. The definitions occur on a new line with the element and colon followed by the syntax.
- Square brackets [ ] enclose optional elements.
- Curly brackets { } enclose choice elements. One element must be chosen among the elements separated by the vertical bars |.
- The parentheses ( ) denote themselves.
- Double hyphens -- denote comments that are not part of the syntax.

CREATE TABLE[11] Syntax

```
CREATE TABLE  TableName
  ( <Column-Definition>*      [   ,      <Table-Constraint>*   ]   )

<Column-Definition>: ColumnName DataType
   [   DEFAULT  {  DefaultValue  |  USER  |  NULL  }  ]
   [   <Embedded-Column-Constraint>+   ]

<Embedded-Column-Constraint>:
   { [  CONSTRAINT  ConstraintName   ]   NOT NULL          |
     [  CONSTRAINT  ConstraintName   ]   UNIQUE            |
     [  CONSTRAINT  ConstraintName   ]   PRIMARY KEY       |
     [  CONSTRAINT  ConstraintName   ]   FOREIGN KEY
          REFERENCES TableName   [   ( ColumnName )   ]
          [  ON DELETE     <Action-Specification>   ]
          [  ON UPDATE     <Action-Specification>   ]   }

<Table-Constraint>: [   CONSTRAINT ConstraintName   ]
     {  <Primary-Key-Constraint>     |
        <Foreign-Key-Constraint>     |
        <Uniqueness-Constraint>  }

<Primary-Key-Constraint>:   PRIMARY KEY ( ColumnName* )

<Foreign-Key-Constraint>:   FOREIGN KEY ( ColumnName* )
       REFERENCES TableName [( ColumnName*   ) ]
       [  ON  DELETE     <Action-Specification>       ]
       [  ON  UPDATE     <Action-Specification>       ]
```

[11] The CHECK constraint, an important kind of table constraint, is described in Chapter 14.

```
<Action-Specification>:  { CASCADE  |  SET NULL  |  SET DEFAULT  |  RESTRICT }

<Uniqueness-Constraint>: UNIQUE ( ColumnName* )
```

Other Related Statements

The ALTER TABLE and DROP TABLE statements support modification of a table definition and deleting a table definition. The ALTER TABLE statement is particularly useful because table definitions often change over time. In both statements, the keyword RESTRICT means that the statement cannot be performed if related tables exist. The keyword CASCADE means that the same action will be performed on related tables.

```
ALTER TABLE  TableName
  { ADD  { <Column-Definition> |  <Table-Constraint>  } |
    ALTER  ColumnName  {  SET DEFAULT DefaultValue  |
             DROP  DEFAULT  }  |
    DROP ColumnName  {  CASCADE  |  RESTRICT } |
    DROP CONSTRAINT ConstraintName  { CASCADE  | RESTRICT  }  }

DROP TABLE TableName  {  CASCADE  |  RESTRICT  }
```

Notes on Oracle Syntax

The CREATE TABLE statement in Oracle 10g SQL conforms closely to the SQL:2003 standard. Here is a list of the most significant syntax differences:

- Oracle SQL does not support the ON UPDATE clause for referential integrity constraints.
- Oracle SQL only supports CASCADE and SET NULL as the action specifications of the ON DELETE clause. If an ON DELETE clause is not specified, the deletion is not allowed (restricted) if related rows exist.
- Oracle SQL does not support dropping columns in the ALTER statement.
- Oracle SQL supports the MODIFY keyword in place of the ALTER keyword in the ALTER TABLE statement (use MODIFY ColumnName instead of ALTER ColumnName).
- Oracle SQL supports data type changes using the MODIFY keyword in the ALTER TABLE statement.

Appendix 3.C

Generation of Unique Values for Primary Keys

The SQL:2003 standard provides the GENERATED clause to support the generation of unique values for selected columns, typically primary keys. The GENERATED clause is used in place of a default value as shown in the following syntax specification. Typically a whole number data type such as INTEGER should be used for columns with a GENERATED clause. The START BY and INCREMENT BY keywords can be used to indicate the initial value and the increment value. The ALWAYS keyword indicates that the

value is always automatically generated. The BY DEFAULT clause allows a user to specify a value, overriding the automatic value generation.

```
<Column-Definition>: ColumnName DataType
    [ <Default-Specification> ]
    [ <Embedded-Column-Constraint>+ ]

<Default-Specification>:
     { DEFAULT { DefaultValue | USER | NULL } |
       GENERATED {ALWAYS | BY DEFAULT } AS IDENTITY
         START WITH NumericConstant
       [ INCREMENT BY NumericConstant ] }
```

Conformance to the SQL:2003 syntax for the GENERATED clause varies among DBMSs. IBM DB2 conforms closely to the syntax. Microsoft SQL Server uses slightly different syntax and only supports the ALWAYS option unless a SET IDENTITY statement is also used. Microsoft Access provides the AutoNumber data type to generate unique values. Oracle uses sequence objects in place of the GENERATED clause. Oracle sequences have similar features except that users must maintain the association between a sequence and a column, a burden not necessary with the SQL:2003 standard.

The following examples contrast the SQL:2003 and Oracle approaches for automatic value generation. Note that the primary key constraint is not required for columns with generated values although generated values are mostly used for primary keys. The Oracle example contains two statements: one for the sequence creation and another for the table creation. Because sequences are not associated with columns, Oracle provides functions that should be used when inserting a row into a table. In contrast, the usage of extra functions is not necessary in SQL:2003.

SQL:2003 GENERATED Clause Example

```
CREATE TABLE Customer
 ( CustNo  INTEGER GENERATED ALWAYS AS IDENTITY
              START WITH  1  INCREMENT BY  1,
...,
CONSTRAINT PKCustomer PRIMARY KEY (CustNo)    )
```

Oracle Sequence Example

```
CREATE SEQUENCE CustNoSeq START WITH 1 INCREMENT BY 1;

CREATE TABLE Customer
( CustNo INTEGER,
...,
CONSTRAINT PKCustomer PRIMARY KEY (CustNo)    ) ;
```