4 Relational Data Model

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 thinking about tables as sets. The combination of familiarity and simplicity with a mathematical foundation is so powerful that relational DBMSs are commercially dominant.

This chapter provides the basic terminology of relational databases and introduces the CREATE TABLE statement of the Structured Query Language (SQL).

4.1 Tables

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.

> Table is a 2D arrangement of data. A table consists of a heading of defining the table name and column names and a body containing rows of data.

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 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 includes a table abbreviation ("Std") followed by a descriptive name. Because column names often are used without identifying the associated tables, the abbreviation supports ease table association. Mixed case highlights the different parts of a column name.

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 on the next page creates the Student table. For each column, the column name and the 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, CHAR for character) and usually a length specification. Table 2 lists common data types used in relational DBMSs.

> Data types are not standard across relational DBMSs. These data types are supported by most systems although the name of the data type may differ.

To create a table for example shown in Table 1., you can use the following SQL statement:

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

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 show relationships between tables. Consider the sample Enrollment table (Table 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 1). For 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 4). Figure 1 shows a graphical depiction of the matching values.

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

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. Similarly, by matching on Enrollment. OfferNo and Offering.OfferNo 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.

When columns have identical names in two tables. it is customary to precede the column name with the table name and a period as Sludent.StdSSN and Enrollment. StdSSN.

You should be aware that other terminology is used besides table, row, and column. Table 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 may hear both "tables" and "fields." You should expect to see a mix of terminology in your career.

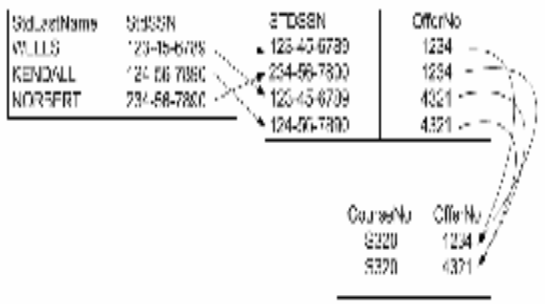

Figure 1 Matching values among the Enrolment, Offering, and Student tables.

4.2 Integrity Rules

Entity integrity (uniqueness integrity) means that each table must have a column or combination of columns with unique values. 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 the database. For auditing and security reasons, it is often important that business entities be easily traceable.

Referential integrity means that the values of columns in one table must match the values of columns in other tables. 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 the database contains valid connections. For example, it is critical that each row of Enrollment contains a social security number of a valid student. Otherwise, some enrollments can be meaningless, possibly resulting in students denied enrollment because non-existing students took their places.

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.

For more precise definitions of entity integrity and referential integrity, a number of other definitions are necessary. These prerequisite definitions and the more precise definitions are presented below.

•**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, as rows in a table must be unique.

•**Candidate key**: a minimal superkey. A superkey is minimal if removing any columns 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 SQL2, 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 columns 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.

Applying 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*, use a *CONSTRAINT* clause for the primary key at the end of the *CREATE TABLE* statement as shown next column. The word *PKStudent* following the *CONSTRAINT* keyword is the name of the constraint.

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

Candidate keys that are not primary keys are declared with the UNIQUE keyword. The *Course* table (see Table 6) contains two candidate keys: *CnurseNo* and *CrsDesc* (course description). The *CourseNo* column is the primary key because it is more stable than the *CrsDesc* column. Course descriptions may change over time, but the course numbers remain the same. In the *CREATE TABLE* statement, a constraint with the keyword *UNIQUE* follows the primary key constraint.

```
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.
Superkeys are usually not important to identify 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.

For referential integrity, the columns *StdSSN* and *OfferNo* are foreign keys in the Enrollment table. The *StdSSN* column refers to Student and the *OfferNo* column refers to the Offering table (Table 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, use *CONSTRAINT* clauses for foreign keys at the end of the *CREATE TABLE* statement:


```
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)
```

Allowing Null Values in Foreign Keys

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 *Enrollrnent.StdSSN* or Enrollment. *OfferNo* because each 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 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 having a related course.

In contrast, the *Offering.FacSSN* column referring to the faculty member teaching the offering may be null. The Faculty table (Table 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 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.

In the *CREATE TABLE* statement, the *NOT NULL* clause indicates that a column cannot have null values. We can specify not allowing NULLS by appending the *NOT NULL* clause after the data type specification. *NOT NULL* clause can also be specified in a *CONSTRAINT* clause.

```
CREATE TABLE Student
(StdSSN CHAR( 11) NOT NULL,
StdFirstName VARCHAR(50) NOT NULL,
StdLastName VARCHAR(50) NOT NULL,
StdCity VARCHAR(50) NOT NULL,
Stdstate CHAR(2) NOT NULL,
StdZipCHAR(10) NOT NULL,
StdMajor CHAR(6),
```

```
StdClass CHAR(2),
StdGPA DECIMA.L(3,2),
CONSTRAINT PKStudent PRIMARY KEY (StdSSN) )
```

Referential Integrity for Self-Referencing (Unary) Relationships

This section finishes with a discussion of self-referencing relationships, a special kind of referential integrity constraint. Self-referencing or unary relationships involve a sin- table. 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.

> 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.

For example, Victoria Emmanuel (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) NOT NULL,
FacFirstName VARCHAR(50) NOT NULL,
FacLastName VARCHAR(50) NOT NULL,
FacCity VARCHAR(50) NOT NULL,
FacState CHAR(2) NOT NULL,
FacZipCode CHAR(10) NOT NULL,
FacHireDate DATE,
FacDept CHAR(6),
FacRank CHAR(4),
FacSalary DECIMAL(10,2),
FacSupervisor CHAR(11),
CONSTRAINT PKFaculty PRIMARY KEY (FacSSN),
CONSTRAINT PKFacSupervisor FOREIGN KEY (FacSupervisor) REFERENCES
Faculty )
```

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 2 (Appendix) 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 "I" table (for example, Student) and the foreign key table (for example. Enrollment) is known as the "M" (many) table.

> **1-M Relationship** a connection between two tables in which one row of a table can be referenced by many rows of a second 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 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 1M 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 2, the linking table *Enrollment* and its relationships with *Offering* and *Student* represent an MN 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 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 fulfill 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 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.

4.3 Delete and update actions for referenced rows

For each referential integrity constraint, you should carefully consider actions on referenced rows. A row is referenced if there is a matching row in a foreign hey table. For example, the first row of the *Course* table (Table 6) with *CourseNo* "IS320" is referenced by the first row of the *Offering* table (Table 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 foreign key table) when the referenced row is deleted?

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

Actions on referenced rows are important when changing the rows of a database. When developing data entry forms, 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.

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**: 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 null 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 added as part of foreign key constraints. For example, the revised *CREATE TABLE* statement for the *Enrollment* table shows *ON DELETE* and *ON UPDATE* actions for the Enrollment table. *NO ACTION* 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 NOT NULL,
StdSSN CHAR(11) NOT NULL,
EnrGrade DECIMAL(3,2),

CONSTRAINT PKEnrollment PRIMARY KEY(OfferNo, StdSSN),

CONSTRAINT FKOfferNo FOREIGN KEY (OfferNo) REFERENCES Offering ON
DELETE NO ACTION ON UPDATE CASCADE,

CONSTRAINT FKStdSSN FOREIGN KEY (StdSSN) REFERENCES Student ON
DELETE NO ACTION 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 adding rows from different tables. When designing data entry forms, we should carefully consider the impact of referential integrity on the order that users complete forms.

4.4 Operations of relational algebra

In previous sections of this chapter, we have familiarized 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.

Restrict (select) and project operators

The *restricts* (also known as *select*) and project operators produce subsets of a table. 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.

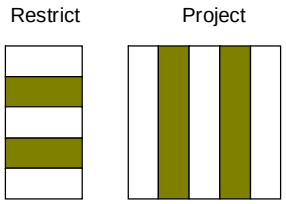

Figure 3 Graphical representation of restrict and project operations.

The *restrict* and *project* operators produce an output table that is a subset of an input table (Figure 3). *Restrict* produces a subset of the rows, while *project* produces a subset of 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.

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 8 shows the result of a restrict operation on Table 4 where the logical expression is: OffDays = 'MW' AND OffTerm = 'SPRING' AND offer = 2000.

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

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

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.

CourseNo
IS320
IS460
IS480

Table 9 Result of a project operation on Offering.CourseNo

For example, if *Offering.CourseNo* is the only column used in a project operation, only three rows are in the result (Table 9) even though the Offering table (Table 4) has nine rows. The column *Offering.CourseNo* contains only three unique values in Table 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 "pure" 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.

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.

The extended cross product (product for short) operator shows everything possible from two tables. The product of two tables is a new table consisting of all possible combinations of rows from the two input tables. Figure 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 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 4, the result table has nine rows and two columns.

Extended Cross Product an operator that builds a table consisting of all possible combinations of rows, from each of the two input tables.

As another example, consider the product of the sample Student (Table 10) and Enrollment (Table 11) tables. The resulting table (Table 12) has nine rows (3 X 3) and seven columns (4 + 3). Note that most rows in the result are not meaningful as only three rows have the same value for *StdSSN*.

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.

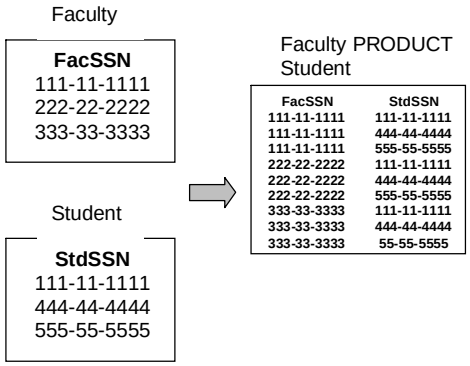

Figure 4 Cross product example

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.

The 'fin 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 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.

The natural join operator, a specialized kind of join, 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. In Figure 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.

As another example, consider the natural join of Student (Table 13) and Enrollment (Table 14) shown in Table 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.

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 5 are *Faculty.FacSSN* and *Offering.FacSSN*.

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

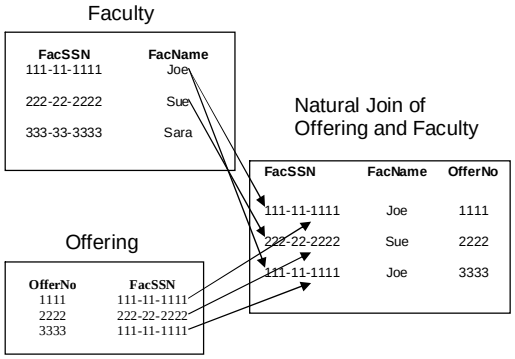

Figure 5 Sample natural join operation

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 15 is the product result shown in Table 12. The second step is to retain only the matching rows (rows 1, 6, and 8 of Table 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 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 5. After learning the join, 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 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.

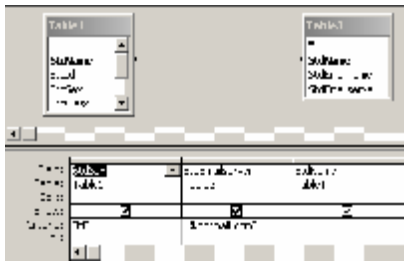

Figure 6 Query Design window showing a join between Student and Enrollment

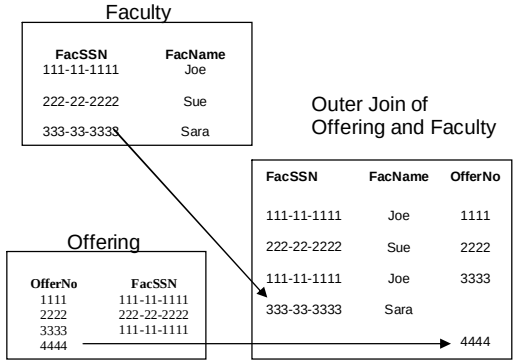

Figure 7 Sample outer join operation

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 non-matching rows. For example, sometimes you 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 non-matching rows in the result as well as to include the matching rows. Figure 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 non-matching rows, null values are used to complete the column values in the other table. In Figure 7 blanks (no values) represent null values. The fourth result row is the non-matched 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 non-matched row of *Offering*.

Full versus One-Sided Outer Join Operators

The outer join operator has two variations. The *full outer join* preserves non-matching rows from both input tables. Figure 7 shows a full *outer join* because the non-matching rows from both tables are preserved in the result. Because it is sometimes useful to preserve the non-matching rows from just one input table, the one-sided *outer Join* operator has been devised. In Figure 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 last row would appear in the result for a one-sided outer join that preserves the rows of the *Offering* table.

Full Outer Join an operator that produces the matching rows (the join part) as well as the non-matching rows from both tables.

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 18, the first two rows are only from the sample *Student* table (Table 16), while the last two rows are only from the sample *Faculty* table (Table 17). Note the use of null values for the columns from the other table. The third row in Table 18 is the row common to the sample *Faculty* and *Student* tables.

One-Sided Outer Join an operator that produces the matching rows (the join part) as well as the non-matching rows from the designated input table.

A one-sided outer join can be useful when a table has null values in a foreign key. For example, the *Offering* table (Table 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 20. With a *natural join*, the first and third rows of Table 20 would not appear.

Visual Formulation of Outer Join Operations

As a query formulation aid, many DBMSs provide a visual way to formulate outer joins. Access provides a visual representation of the one-sided join operator in the Query Design window. Figure 8 depicts a one-sided outer join that preserves the rows of the Offering table. The arrow from Offering to Faculty means that the non-matched 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 non-matched rows of Faculty, and (3) show matched rows and non-matched rows of Offering. Choice (3) is shown in Figure 8. Choice (1) would appear similar to Figure 6. Choice (2) would have the arrow from Faculty to Offering.

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

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.

Figure 8 Query design window showing a one-sided outer join

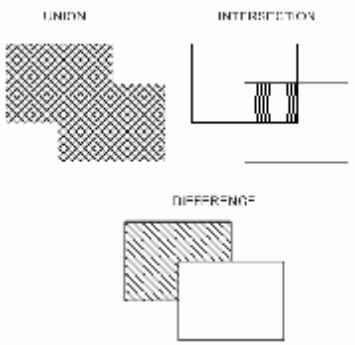

Figure 9 Venn diagrams for traditional set operators

Union Compatibility

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 compatible means that each table must have the same number of columns and each corresponding column must have the same data type. Union compatibility can be confusing because it requires positional correspondence of the columns. That is. the first columns of the two tables must have the same data type, the second columns must have the same data type, and so on.

To depict the union, intersection, and difference operators, let us apply them to the *Student1* and *Student2* tables (Tables 21 and 22). These tables are union compatible because they have identical columns listed in the same order. The results of union, intersection, and difference operators are shown in Tables 23 through 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.

Union Compatibility a requirement on the input tables for the traditional set operators. Both tables must have the same number of columns and each corresponding column must have the same data type.

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 do have some important, specialized uses. One use is to combine tables distributed over many 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, use *UBSStudent UNION BSUStudent*. To find students only attending Big State, use *BSUStudent DIFFERENCE UBSStudent*. To find students attending both universities, 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 identical 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 discussed in Section "**Restrict (select) and project operators".**

Summarize Operator

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 calculated values. Figure 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 27 shows the result of a summarize operation on the sample Faculty table in Table 26. Note that the result contains one row per value of the grouping column, *FacDept*.

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.

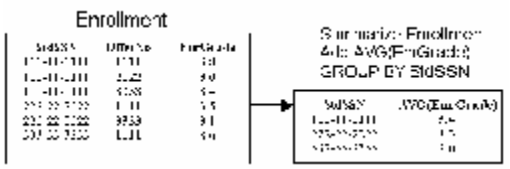

Figure 10 Sample summarize operation

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.

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 can be more difficult to understand. When appropriate, the divide operator provides a powerful way to combine tables.

> **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 the second table.

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 contains 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 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). The result table has one column where the values come from the first column of the binary table. The result table in Figure 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 division operator in another way, rewrite the SuppPart table as three rows using the angle brackets <> to surround a row: <s3, {pl, p2, p3}>, <s0, {p1

}>, <s1, {p2}>. Rewrite the Part table as a set: {p1, p2}. The value s3 is in the result table because its set of second column values {pl, p2, p3 } contains the values in the second table {p1, p2}. The other *SuppNo* values (s0 and s1) are not in the result because they are not associated with all the values in the *Part* table.

As an example using the university database tables, Table 30 shows the result of a divide operation involving the sample *Enrollment* (Table 28) and *Student* tables (Table 29). The result shows offerings in which every student is enrolled. Only *OfferNo 4235* has all three students enrolled.

Summary of Operators

To help you recall the relational algebra operators, Tables 31 and 32 provide a convenient summary of the meaning and usage of each operator.

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

SuppPart

SuppNo	PartNo
s3	p1
s3	p2
s3	p3
s0	p1
s1	p2

Part

PartNo
p1
p2

SuppPart DIVIDEBY Part

SuppNo
s3

s3 {p1,p2,p3} contains {p1,p2}

Figure 11 Sample divide operation

APPENDIX

Table 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	KNEDALL	TACOMA	WA	99042-3321	ACCT	JR	3.50

Table 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 as for CHAR columns. Most DBMSs have an upper limit on the length such as 255.
FLOAT(P)	F or columns containing numeric data with 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.

Table 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 4 Sample Offering Table

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	FacSSN	OffDays
1111	IS320	SUMMER	2000	BLM302	10:30 AM		MW
1234	IS320	FALL	1999	BLM302	10:30 AM	098-76-5432	MW
2222	IS460	SUMMER	1999	BLM412	1:30 PM		TTH
3333	IS320	SPRING	2000	BLM214	8:30 AM	098-76-5432	MW
4321	IS320	FALL	1999	BLM214	3:30 PM	098-76-5432	TTH
4444	IS320	SPRING	2000	BLM302	3:30 PM	543-21-0987	TTH
5678	IS480	SPRING	2000	BLM302	10:30 AM	987-65-4321	MW
5679	IS480	SPRING	2000	BLM412	3:30 PM	876-54-3210	TTH
9876	IS460	SPRING	2000	BLM307	1:30 PM	654-32-1098	TTH

Table 5 alternative Terminology for Relational Databases

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

Table 6 Sample Course Table

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

Table 7 Sample Faculty Table

FacSSN	FacFirst Name	FacLastName	FacCity	FacState	FacDept	FacRank	FacSalary	FacSupervisor	FacHire Date	FacZipCode
098-76-5432	LEONARD	VINCE	SEATTLE	WA	MS	ASST	$35,000	654-32-1098	0l-Apr-90	98111-9921
543-21-0987	VICTORIA	EMMANUEL	BOTHELL	WA	MS	PROF	$120.000		01-Apr-91	98011-2242
654-32-1098	LEONARD	FIBON	SEATTLE	WA	NIS	ASSC	$70.000	5.13-21-0987	01-Apr-89	98121-0094
765-43-2109	VICKI	MACON	BELLEVUE	WA	FIN	PROF	$65,000		01-Apr-92	98015-9945
876-54-3210	CHRIS TOPHER	COLAN	SEATTLE	WA	MS	ASST	$40.000	651-32-1098	01-Apr-94	98114-1332
987-65-4321	JULIA	MILLS	SEATTL.E	WA	PIN	ASSC	$75,000	765-43-2109	01-Apr-95	98114-9954

Table 8 Result of restrict operation on the sample offering table (Table 4)

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	EacSSN	OffDay
3333	IS320	SPRING	2000	BLM214	8:30 AM	098-76-5432	MW
5678	IS480	SPRING	2000	BLM302	10:30 AM	987-65-4321	MW

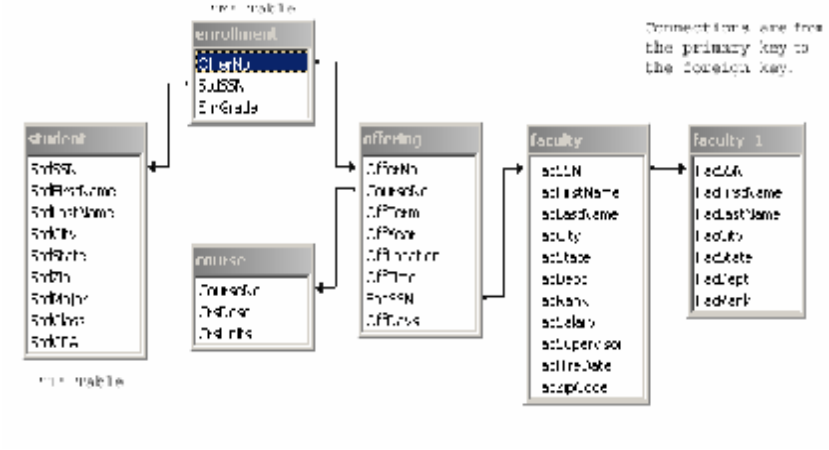

Figure 2 Relationship window for the university database.

Table 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 11 Sample Enrollment Table

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

Table 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

Table 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 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 15 Natural join of Student and Enrollment

Student.StdSSN	StdLastName	StdMajor	FRStdClass	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

Table 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 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 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 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 20 Result of one-sided outer join between Offering (Table 19) and Faculty (Table 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

Table 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 22 Student2 table

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

Table 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 24 Student1 INTERESCT Student2

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

Table 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

Table 26 Sample faculty table

FacSSN	FacLastName	FacDept	FacRank	FacSalary	FacSupervisor	FacHireDate
098-76-5432	VINCE	MS	ASST	35000	654-32-1098	01-Apr-90
543-21-0987	EMMANUEL	MS	PROF	120000		01-Apr-91
654-32-1098	FIBON	MS	ASSC	70000	543-21-0987	01-Apr-89
765-43-2109	MACON	FIN	PROF	65000		01-Apr-92
876-54-3210	COLAN	MS	ASST	40000	654-32-1098	01-Apr-94
987-65-4321	MILLS	FIN	ASSC	75000	765-43-2109	01-Apr-95

Table 27 Result table for SUMMARIZE Faculty ADD AVG(FacSalary) GROUP BY FacDept

FacDept	FacSalary
MS	66250
FIN	70000

Table 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 29 Sample Student table

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

Table 30 Result of enrollment DIVIDEBY Student

OfferNo
4235

Table 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 (2-column) table that match (in the other column) all values in a unary (l-column) table.
Summarize	Organizes a table on specified grouping columns. Specified aggregate computations are made on each value of the grouping columns.

Table 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).