# UNIT – 2

# RELATIONAL MODEL AND RELATIONAL ALGEBRA

## 2.1 Relational Model concepts

The relational model represents the database as a collection of *relations.* Informally, each relation resembles a table of values or, to some extent, a *flat* file of records. It is called a **flat file** because each record has a simple linear or *flat* structure. For example, the database of files that are shown in Figure 2.1 is similar to the basic relational model representation.

**STUDENT**

| Name | Student_number | Class | Major |
|---|---|---|---|
| Smith | 17 | 1 | CS |
| Brown | 8 | 2 | CS |

**COURSE**

| Course_name | Course_number | Credit_hours | Department |
|---|---|---|---|
| Intro to Computer Science | CS1310 | 4 | CS |
| Data Structures | CS3320 | 4 | CS |
| Discrete Mathematics | MATH2410 | 3 | MATH |
| Database | CS3380 | 3 | CS |

**SECTION**

| Section_identifier | Course_number | Semester | Year | Instructor |
|---|---|---|---|---|
| 85 | MATH2410 | Fall | 07 | King |
| 92 | CS1310 | Fall | 07 | Anderson |
| 102 | CS3320 | Spring | 08 | Knuth |
| 112 | MATH2410 | Fall | 08 | Chang |
| 119 | CS1310 | Fall | 08 | Anderson |
| 135 | CS3380 | Fall | 08 | Stone |

**GRADE_REPORT**

| Student_number | Section_identifier | Grade |
|---|---|---|
| 17 | 112 | B |
| 17 | 119 | C |
| 8 | 85 | A |
| 8 | 92 | A |
| 8 | 102 | B |
| 8 | 135 | A |

**Figure 2.1** A database that stores student and course information.

When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a realworld entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row.

For example, the first table of Figure 2.1 is called STUDENT because each row represents facts about a particular student entity. The column names—Name, Student_number, Class, and Major—specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a *tuple,* a column header is called an *attribute,* and the table is called a *relation.* The data type describing the types of values that can appear in each column is represented by a *domain* of possible values. We now define these terms—*domain, tuple, attribute,* and *relation*— formally.

### 2.1.1 Domains, Attributes, Tuples, and Relations

A **domain** *D* is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values.

A **relation schema** R, denoted by *R* (*A*1, *A*2... An), is made up of a relation name *R* and a list of attributes, *A*1, *A*2... *An*. Each **attribute** *Ai* is the name of a role played by some domain *D* in the relation schema *R*. *D* is called the **domain** of *Ai* and is denoted by **dom**(**A**i*).* A relation schema is used to *describe* a relation; *R* is called the **name** of this relation. The **degree** (or **arity**) of a relation is the number of attributes *n* of its relation schema.

A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:
STUDENT (Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)
Using the data type of each attribute, the definition is sometimes written as:
STUDENT (Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)

More precisely, we can specify the following previously defined domains for some of the attributes of the STUDENT relation: dom(Name) = Names; dom(Ssn) = Social_security_numbers; dom(HomePhone) = USA_phone_numbers2 and so on.

A **relation** (or **relation state**) *r* of the relation schema *R* (*A*1, *A*2... *An*), also denoted by *r*(*R*), is a set of *n*-tuples *r* = {*t*1, *t*2... *tm*}. Each ***n*-tuple** *t* is an ordered list of *n* values *t* =<*v*1*, v*2... *vn*>, where each value *vi,* 1 ≤ *i* ≤ *n*, is an element of dom *(Ai*). The *i*th value in tuple *t*, which corresponds to the attribute *Ai*, is referred to as *t* [*Ai*].

Figure 2.2 shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity (or object). We display the relation as a table, where each tuple is shown as a *row* and each attribute corresponds to a *column header* indicating a role or interpretation of the values in that column. *NULL values* represent attributes whose values are unknown or do not exist for some individual STUDENT tuple.

Relation Name

Attributes

Tuples

**STUDENT**

| Name | Ssn | Home_phone | Address | Office_phone | Age | Gpa |
|---|---|---|---|---|---|---|
| Benjamin Bayer | 305-61-2435 | (817)373-1616 | 2918 Bluebonnet Lane | NULL | 19 | 3.21 |
| Chung-cha Kim | 381-62-1245 | (817)375-4409 | 125 Kirby Road | NULL | 18 | 2.89 |
| Dick Davidson | 422-11-2320 | NULL | 3452 Elgin Road | (817)749-1253 | 25 | 3.53 |
| Rohan Panchal | 489-22-1100 | (817)376-9821 | 265 Lark Lane | (817)749-6492 | 28 | 3.93 |
| Barbara Benson | 533-69-1238 | (817)839-8461 | 7384 Fontana Lane | NULL | 19 | 3.25 |

**Figure 2.2** The attributes and tuples of a relation STUDENT.

The earlier definition of a relation can be *restated* more formally using set theory concepts as follows. A relation (or relation state) $r(R)$ is a **mathematical relation** of degree $n$ on the domains dom($A1$), dom($A2$), ..., dom($An$), which is a **subset** of the **Cartesian product** (denoted by ×) of the domains that define $R$:

$$r(R) \subseteq (\mathrm{dom}(A_1) \times \mathrm{dom}(A_2) \times \ldots \times \mathrm{dom}(A_n))$$

The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the total number of values, or **cardinality,** in a domain $D$ by $|D|$ (assuming that all domains are finite), the total number of tuples in the Cartesian product is

$$|\mathrm{dom}(A_1)| \times |\mathrm{dom}(A_2)| \times \ldots \times |\mathrm{dom}(A_n)|$$

### 2.1.2 Characteristics of Relations

#### Ordering of Tuples in a Relation

A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. In other words, a relation is not sensitive to the ordering of tuples. When we display a relation as a table, the rows are displayed in a certain order. Many tuple orders can be specified on the same relation. For example, tuples in the STUDENT relation in Figure 2.3 could be ordered by values of Name, Ssn, Age, or some other attribute. The definition of a relation does not specify any order: There is *no preference* for one ordering over another. Hence, the relation displayed in Figure 2.2 is considered *identical* to the one shown in Figure 2.3.

**STUDENT**

| Name | Ssn | Home_phone | Address | Office_phone | Age | Gpa |
|---|---|---|---|---|---|---|
| Dick Davidson | 422-11-2320 | NULL | 3452 Elgin Road | (817)749-1253 | 25 | 3.53 |
| Barbara Benson | 533-69-1238 | (817)839-8461 | 7384 Fontana Lane | NULL | 19 | 3.25 |
| Rohan Panchal | 489-22-1100 | (817)376-9821 | 265 Lark Lane | (817)749-6492 | 28 | 3.93 |
| Chung-cha Kim | 381-62-1245 | (817)375-4409 | 125 Kirby Road | NULL | 18 | 2.89 |
| Benjamin Bayer | 305-61-2435 | (817)373-1616 | 2918 Bluebonnet Lane | NULL | 19 | 3.21 |

**Figure 2.3** The relation STUDENT from Figure 2.1 with a different order of tuples.

### Ordering of Values within a Tuple and an Alternative Definition of a Relation

According to the preceding definition of a relation, an *n*-tuple is an *ordered list* of *n* values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important. However, at a more abstract level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained. According to this definition of tuple as a mapping, a **tuple** can be considered as a **set** of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute *Ai* to a value *vi* from dom(*Ai*). The ordering of attributes is *not* important, because the *attribute name* appears with its *value*. By this definition, the two tuples shown in Figure 2.4 are identical.

*t* = < (Name, Dick Davidson),(Ssn, 422-11-2320),(Home_phone, NULL),(Address, 3452 Elgin Road),
(Office_phone, (817)749-1253),(Age, 25),(Gpa, 3.53)>

*t* = < (Address, 3452 Elgin Road),(Name, Dick Davidson),(Ssn, 422-11-2320),(Age, 25),
(Office_phone, (817)749-1253),(Gpa, 3.53),(Home_phone, NULL)>

**Figure 2.4** Two identical tuples when the order of attributes and values is not part of relation definition.

### Values and NULLs in the Tuples

Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes are not allowed. This model is sometimes called the **flat relational model**. Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption. Hence, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes in the basic relational model. An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases. For example, in Figure 2.2, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone *does not apply* to these students). In general, we can have several meanings for NULL values, such as ***value unknown, value*** exists but is ***not available***, or ***attribute does not apply*** to this tuple *(*also known as ***value undefined****).*

**Interpretation (Meaning) of a Relation**

The relation schema can be interpreted as a declaration or a type of **assertion**. For example, the schema of the STUDENT relation of Figure 2.2 asserts that, in general, a student entity has a Name, Ssn, Home_phone, Address, Office_phone, Age, and Gpa. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion.

For example, the first tuple in Figure 2.2 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 205-61-2425, Age is 19, and so on. Notice that some relations may represent facts about *entities,* whereas other relations may represent facts about *relationships.* For example, a relation schema MAJORS (Student_ssn, Department_code) asserts that students major in academic disciplines.A tuple in this relation relates a student to his or her major discipline. Hence, the relational model represents facts about both entities and relationships *uniformly* as relations. This sometimes compromises understandability because one has to guess whether a relation represents an entity type or a relationship type.

**2.1.3 Relational Model Notation**

We will use the following notation in our presentation:
- A relation schema $R$ of degree $n$ is denoted by $R(A1, A2, ..., An)$.
- The uppercase letters $Q$, $R$, $S$ denote relation names.
- The lowercase letters $q$, $r$, $s$ denote relation states.
- The letters $t$, $u$, $v$ denote tuples.
- In general, the name of a relation schema such as STUDENT also indicates the current set of tuples in that relation—the *current relation state*—whereas STUDENT (Name, Ssn, ...) refers *only* to the relation schema.
- An attribute $A$ can be qualified with the relation name $R$ to which it belongs by using the dot notation $R.A$—for example, STUDENT.Name or STUDENT.Age. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.
- An $n$-tuple $t$ in a relation $r(R)$ is denoted by $t = <v1, v2... vn>$, where $vi$ is the value corresponding to attribute $Ai$. The following notation refers to **component values** of tuples:
- Both $t[Ai]$ and $t.Ai$ (and sometimes $t[i]$) refer to the value $vi$ in $t$ for attribute $Ai$.
- Both $t[Au, Aw, ..., Az]$ and $t.(Au, Aw, ..., Az)$, where $Au, Aw, ..., Az$ is a list of attributes from $R$, refer to the subtuple of values $<vu, vw, ..., vz>$ from $t$ corresponding to the attributes specified in the list.

**2.2 Relational Model Constraints and Relational Database Schemas**

In a relational database, there will typically be many relations, and the tuples in those relations are usually related in various ways. The state of the whole database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or **constraints** on the actual values in a database state. In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints.

Constraints on databases can generally be divided into three main categories:

1. Constraints that is inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.
2. Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the DDL. We call these as **schema-based constraints** or **explicit constraints**.
2. Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs. We call these **application-based** or **semantic constraints** or **business rules**.

### 2.2.1 Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute *A* must be an atomic value from the domain dom(*A*). The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and doubleprecision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as is date, time, timestamp, and money, or other special data types. Other possible domains may be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed.

### 2.2.2 Key Constraints and Constraints on NULL Values

In the formal relational model, a *relation* is defined as a *set of tuples.* By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema *R* with the property that no two tuples in any relation state *r* of *R* should have the same combination of values for these attributes.

Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key,* which has no redundancy. A **key** *K* of a relation schema *R* is a superkey of *R* with the additional property that removing any attribute *A* from *K* leaves a set of attributes *K_* that is not a superkey of *R* any more.

Hence, a key satisfies two properties:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to a super key.
2. It is a *minimal super key*—that is, a super key from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold. This property is not required by a super key.

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation in Figure 2.5 has two candidate keys: License_number and Engine_serial_number. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation.We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Figure 2.5. Notice that when a relation schema has several candidate keys,

**CAR**

| License_number | Engine_serial_number | Make | Model | Year |
|---|---|---|---|---|
| Texas ABC-739 | A69352 | Ford | Mustang | 02 |
| Florida TVP-347 | B43696 | Oldsmobile | Cutlass | 05 |
| New York MPO-22 | X83554 | Oldsmobile | Delta | 01 |
| California 432-TFY | C43742 | Mercedes | 190-D | 99 |
| California RSK-629 | Y82935 | Toyota | Camry | 04 |
| Texas RSK-629 | U028365 | Jaguar | XJS | 04 |

**Figure 2.5** The CAR relation, with two candidate keys: License_number and Engine_serial_number.

the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as **unique keys**, and are not underlined. Another constraint on attributes specifies whether NULL values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

### 2.2.3 Relational Databases and Relational Database Schemas

A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section we define a relational database and a relational database schema.

A **relational database schema** *S* is a set of relation schemas S = {*R*1, *R*2... *Rm*}and a set of **integrity constraints** IC. A **relational database state** DB of *S* is a set of relation states DB = {*r*1, *r*2, *rm*} such that each *ri* is a state of *Ri* and such that the *ri* relation states satisfy the integrity constraints specified in IC. Figure 2.6 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. The underlined attributes represent primary keys. Figure 2.7 shows a relational database state corresponding to the COMPANY schema. When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is called an **invalid state**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state.**

**EMPLOYEE**

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|---|---|---|---|---|---|---|---|---|---|

**DEPARTMENT**

| Dname | Dnumber | Mgr_ssn | Mgr_start_date |
|---|---|---|---|

**DEPT_LOCATIONS**

| Dnumber | Dlocation |
|---|---|

**PROJECT**

| Pname | Pnumber | Plocation | Dnum |
|---|---|---|---|

**WORKS_ON**

| Essn | Pno | Hours |
|---|---|---|

**DEPENDENT**

| Essn | Dependent_name | Sex | Bdate | Relationship |
|---|---|---|---|---|

**Figure 2.6** Schema diagram for the COMPANY relational database schema.

**EMPLOYEE**

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|---|---|---|---|---|---|---|---|---|---|
| John | B | Smith | 123456789 | 1965-01-09 | 731 Fondren, Houston, TX | M | 30000 | 333445555 | 5 |
| Franklin | T | Wong | 333445555 | 1955-12-08 | 638 Voss, Houston, TX | M | 40000 | 888665555 | 5 |
| Alicia | J | Zelaya | 999887777 | 1968-01-19 | 3321 Castle, Spring, TX | F | 25000 | 987654321 | 4 |
| Jennifer | S | Wallace | 987654321 | 1941-06-20 | 291 Berry, Bellaire, TX | F | 43000 | 888665555 | 4 |
| Ramesh | K | Narayan | 666884444 | 1962-09-15 | 975 Fire Oak, Humble, TX | M | 38000 | 333445555 | 5 |
| Joyce | A | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX | F | 25000 | 333445555 | 5 |
| Ahmad | V | Jabbar | 987987987 | 1969-03-29 | 980 Dallas, Houston, TX | M | 25000 | 987654321 | 4 |
| James | E | Borg | 888665555 | 1937-11-10 | 450 Stone, Houston, TX | M | 55000 | NULL | 1 |

**DEPARTMENT**

| Dname | Dnumber | Mgr_ssn | Mgr_start_date |
|---|---|---|---|
| Research | 5 | 333445555 | 1988-05-22 |
| Administration | 4 | 987654321 | 1995-01-01 |
| Headquarters | 1 | 888665555 | 1981-06-19 |

**DEPT_LOCATIONS**

| Dnumber | Dlocation |
|---|---|
| 1 | Houston |
| 4 | Stafford |
| 5 | Bellaire |
| 5 | Sugarland |
| 5 | Houston |

**WORKS_ON**

| Essn | Pno | Hours |
|---|---|---|
| 123456789 | 1 | 32.5 |
| 123456789 | 2 | 7.5 |
| 666884444 | 3 | 40.0 |
| 453453453 | 1 | 20.0 |
| 453453453 | 2 | 20.0 |
| 333445555 | 2 | 10.0 |
| 333445555 | 3 | 10.0 |
| 333445555 | 10 | 10.0 |
| 333445555 | 20 | 10.0 |
| 999887777 | 30 | 30.0 |
| 999887777 | 10 | 10.0 |
| 987987987 | 10 | 35.0 |
| 987987987 | 30 | 5.0 |
| 987654321 | 30 | 20.0 |
| 987654321 | 20 | 15.0 |
| 888665555 | 20 | NULL |

**PROJECT**

| Pname | Pnumber | Plocation | Dnum |
|---|---|---|---|
| ProductX | 1 | Bellaire | 5 |
| ProductY | 2 | Sugarland | 5 |
| ProductZ | 3 | Houston | 5 |
| Computerization | 10 | Stafford | 4 |
| Reorganization | 20 | Houston | 1 |
| Newbenefits | 30 | Stafford | 4 |

**DEPENDENT**

| Essn | Dependent_name | Sex | Bdate | Relationship |
|---|---|---|---|---|
| 333445555 | Alice | F | 1986-04-05 | Daughter |
| 333445555 | Theodore | M | 1983-10-25 | Son |
| 333445555 | Joy | F | 1958-05-03 | Spouse |
| 987654321 | Abner | M | 1942-02-28 | Spouse |
| 123456789 | Michael | M | 1988-01-04 | Son |
| 123456789 | Alice | F | 1988-12-30 | Daughter |
| 123456789 | Elizabeth | F | 1967-05-05 | Spouse |

**Figure 2.7** One possible database state for the COMPANY relational database schema.

### 2.2.4 Integrity, Referential Integrity and Foreign Keys

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation.

To define referential integrity more formally, first we define the concept of a *foreign key.* The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas *R*1 and *R*2. A set of attributes FK in relation schema *R*1 is a **foreign key** of *R*1 that **references** relation *R*2 if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of *R*2; the attributes FK are said to **reference** or **refer to** the relation *R*2.
2. A value of FK in a tuple *t*1 of the current state *r*1(*R*1) either occurs as a value of PK for some tuple *t*2 in the current state *r*2(*R*2) *or is NULL*. In the former case, we have *t*1[FK] = *t*2[PK], and we say that the tuple *t*1 **references** or **refers to** the tuple *t*2.

In this definition, *R*1 is called the **referencing relation** and *R*2 is the **referenced relation**. If these two conditions hold, a **referential integrity constraint** from *R*1 to *R*2 is said to hold. For example, consider the database shown in Figure 2.7. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation.

Figure 2.8 shows the schema in Figure 2.6 with the referential integrity constraints displayed in this manner. The DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. Most relational DBMSs support key, entity integrity, and referential integrity constraints. These constraints are specified as a part of data definition in the DDL.

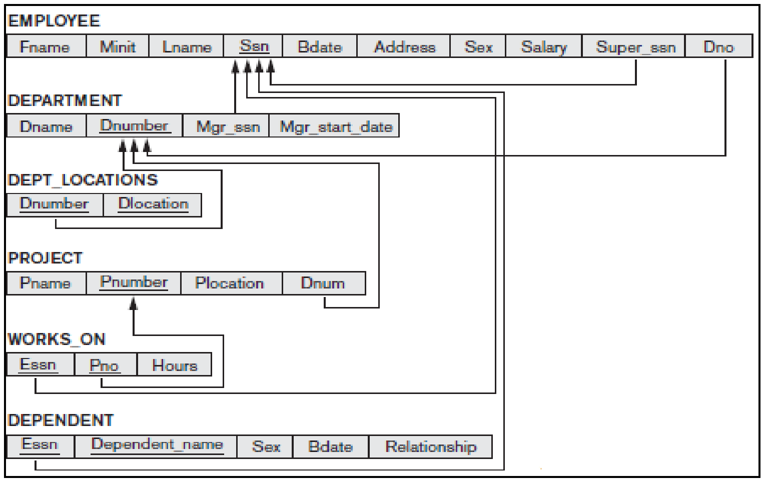


**Figure 2.8** Referential integrity constraints displayed on the COMPANY relational database schema.

### 2.2.5 Other Types of Constraints

The preceding integrity constraints are included in the data definition language because they occur in most database applications. However, they do not include a large class of general constraints, sometimes called *semantic integrity constraints,* which may have to be specified and enforced on a relational database. Examples of such constraints are *the salary of an employee should not exceed the salary of the employee's supervisor* and *the maximum number of hours an employee can work on all projects per week is 56*. Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language**. Mechanisms called **triggers** and **assertions** can be used.

Another type of constraint is the *functional dependency* constraint, which establishes a functional relationship among two sets of attributes *X* and *Y*. This constraint specifies that the value of *X* determines a unique value of *Y* in all states of a relation; it is denoted as a functional dependency $X \rightarrow Y$.

The types of constraints we discussed so far may be called **state constraints** because they define the constraints that a *valid state* of the database must satisfy. Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database.

## 2.3 Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates.* The relational algebra operations can be used to specify **retrievals**. A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database to retrieve information. The user formulates a query that specifies the data of interest, and a new relation is formed by applying relational operators to retrieve this data. The **result relation** becomes the answer to (or result of) the user's query.

In this section, we concentrate on the database **modification** or **update** operations. There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records. Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss the types of constraints that may be violated by each of these operations and the types of actions that may be taken if an operation causes a violation.

### 2.3.1 The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple *t* that is to be inserted into a relation *R*. Insert can violate any of the four types of constraints discussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. Key constraints can be violated if a key value in the new tuple *t* already exists in another tuple in the relation *r*(*R*). Entity integrity can be violated if any part of the primary key of the new tuple *t* is NULL. Referential integrity can be violated if the value of any foreign key in *t* refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion using Figure 2.7.

■ *Operation*: Insert <'Cecilia', 'F', 'Kolonsky', NULL, '1960-04-05', '6257 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.
*Result*: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.

■ *Operation*: Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6257 Windy Lane, Katy, TX', F, 28000, '987654221', 4> into EMPLOYEE.
*Result*: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.

■ *Operation*: Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6257 Windswept, Katy, TX', F, 28000, '987654221', 7> into EMPLOYEE.
*Result*: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

■ *Operation*: Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6257 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.
*Result*: This insertion satisfies all constraints, so it is acceptable.

### 2.3.2 The Delete Operation

The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.

■ *Operation*: Delete the WORKS_ON tuple with Essn = '999887777' and Pno = 10.
*Result*: This deletion is acceptable and deletes exactly one tuple.

■ *Operation*: Delete the EMPLOYEE tuple with Ssn = '999887777'.
*Result*: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

■ *Operation*: Delete the EMPLOYEE tuple with Ssn = '222445555'.
*Result*: This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT,WORKS_ON, and DEPENDENT relations.

### 2.3.3 The Update Operation

The **Update** (or **Modify**) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

■ *Operation*: Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.
*Result*: Acceptable.

■ *Operation*: Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.
*Result*: Acceptable.

■ *Operation*: Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.
*Result*: Unacceptable, because it violates referential integrity.

■ *Operation*: Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654221'.
*Result*: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.

### 2.3.4 The Transaction Concept

A database application program running against a relational database typically executes one or more *transactions*. A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database. At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema. A single transaction may involve any number of retrieval operations, any number of update operations. These retrievals and updates will together form an atomic unit of work against the database.

For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount. A large number of commercial applications running against relational databases in **online transaction processing (OLTP**) systems are executing transactions at rates that reach several hundred per second.

## 2.4 Unary Relational Operations: SELECT and PROJECT

### 2.4.1 The SELECT Operation

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**. One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded.

For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $20,000, we can individually specify each of these two conditions with a SELECT operation as follows:

$$\sigma_{\text{Dno}=4}(\text{EMPLOYEE})$$
$$\sigma_{\text{Salary}>30000}(\text{EMPLOYEE})$$

In general, the SELECT operation is denoted by

$$\sigma_{<\text{selection condition}>}(R)$$

where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation *R*. Notice that *R* is generally a *relational algebra expression* whose result is a relation—the simplest such expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as *R*.

The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value>
or
<attribute name> <comparison op> <attribute name>

where <attribute name> is the name of an attribute of *R*, <comparison op> is normally one of the operators {=, <, ≤, >, ≥, ≠} and <constant value> is a constant value from the attribute domain. Clauses can be connected by the standard Boolean operators *and*, *or*, and *not* to form a general selection condition. For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $20,000, we can specify the following SELECT operation:

$$\sigma_{(\text{Dno}=4\ \textbf{AND}\ \text{Salary}>25000)\ \textbf{OR}\ (\text{Dno}=5\ \textbf{AND}\ \text{Salary}>30000)}(\text{EMPLOYEE})$$

The result is shown in Figure 2.9 (a).

**(a)**

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|---|---|---|---|---|---|---|---|---|---|
| Franklin | T | Wong | 333445555 | 1955-12-08 | 638 Voss, Houston, TX | M | 40000 | 888665555 | 5 |
| Jennifer | S | Wallace | 987654321 | 1941-06-20 | 291 Berry, Bellaire, TX | F | 43000 | 888665555 | 4 |
| Ramesh | K | Narayan | 666884444 | 1962-09-15 | 975 Fire Oak, Humble, TX | M | 38000 | 333445555 | 5 |

**(b)**

| Lname | Fname | Salary |
|---|---|---|
| Smith | John | 30000 |
| Wong | Franklin | 40000 |
| Zelaya | Alicia | 25000 |
| Wallace | Jennifer | 43000 |
| Narayan | Ramesh | 38000 |
| English | Joyce | 25000 |
| Jabbar | Ahmad | 25000 |
| Borg | James | 55000 |

**(c)**

| Sex | Salary |
|---|---|
| M | 30000 |
| M | 40000 |
| F | 25000 |
| F | 43000 |
| M | 38000 |
| M | 25000 |
| M | 55000 |

**Figure 2.9**
Results of SELECT and PROJECT operations. (a) $\sigma_{(\text{Dno}=4\ \textbf{AND}\ \text{Salary}>25000)\ \text{OR}\ (\text{Dno}=5\ \textbf{AND}\ \text{Salary}>30000)}$ (EMPLOYEE). (b) $\pi_{\text{Lname, Fname, Salary}}$(EMPLOYEE). (c) $\pi_{\text{Sex, Salary}}$(EMPLOYEE).

Notice that all the comparison operators in the set {=, <, ≤, >, ≥, ≠} can apply to attributes whose domains are *ordered values*, such as numeric or date domains. An example of an unordered domain is the domain Color = { 'red', 'blue', 'green', 'white', 'yellow'}, where no order is specified among the various colors. Some domains allow additional types of comparison operators; for example, a domain of character strings may allow the comparison operator **SUBSTRING_OF**.

The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:

- (cond1 **AND** cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
- (cond1 **OR** cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
- (**NOT** cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

The SELECT operator is **unary**; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually*; hence, selection conditions cannot involve more than one tuple. The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

Notice that the SELECT operation is **commutative**; that is,

$$\sigma_{<\text{cond1}>}(\sigma_{<\text{cond2}>}(R)) = \sigma_{<\text{cond2}>}(\sigma_{<\text{cond1}>}(R))$$

Hence, a sequence of SELECTs can be applied in any order. In addition, we can always combine a **cascade** (or **sequence**) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is,

$$\sigma_{<\text{cond1}>}(\sigma_{<\text{cond2}>}(\ldots(\sigma_{<\text{cond}n>}(R))\ \ldots)) = \sigma_{<\text{cond1}>\ \textbf{AND}<\text{cond2}>\ \textbf{AND}\ldots\textbf{AND}\ <\text{cond}n>}(R)$$

In SQL, the SELECT condition is typically specified in the WHERE clause of a query. For example, the following operation:

$$\sigma_{\text{Dno}=4\ \textbf{AND}\ \text{Salary}>25000}\ (\text{EMPLOYEE})$$

would correspond to the following SQL query:

```
SELECT    *
FROM      EMPLOYEE
WHERE     Dno=4 AND Salary>25000;
```

### 2.4.2 The PROJECT Operation

If we think of a relation as a table, the SELECT operation chooses some of the *rows* from the table while discarding other rows. The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns i.e. attributes. Therefore, the result of the PROJECT operation can be visualized as a *vertical partition* of the relation into two relations: one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns. For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{\text{Lname, Fname, Salary}}(\text{EMPLOYEE})$$

The resulting relation is shown in Figure 2.9(b). The general form of the PROJECT operation is

$$\pi_{<\text{attribute list}>}(R)$$

where π(pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired sublist of attributes from the attributes of relation *R*. Again, notice that *R* is, in general, a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attributes specified in <attribute list> *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in <attribute list>.

If the attribute list includes only nonkey attributes of *R*, duplicate tuples are likely to occur. The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**.

In SQL, the PROJECT attribute list is specified in the SELECT clause of a query. For example, the following operation:

$\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$

would correspond to the following SQL query:

```
SELECT   DISTINCT Sex, Salary
FROM     EMPLOYEE
```

Notice that if we remove the keyword **DISTINCT** from this SQL query, then duplicates will not be eliminated. This option is not available in the formal relational algebra.

### 2.4.3 Sequences of Operations and the RENAME Operation

In general, for most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results.

For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression, also known as an **in-line expression**, as follows:

$$\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$$

Figure 2.10(a) shows the result of this in-line relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$$

(a)

| Fname | Lname | Salary |
|---|---|---|
| John | Smith | 30000 |
| Franklin | Wong | 40000 |
| Ramesh | Narayan | 38000 |
| Joyce | English | 25000 |

(b)

TEMP

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|---|---|---|---|---|---|---|---|---|---|
| John | B | Smith | 123456789 | 1965-01-09 | 731 Fondren, Houston,TX | M | 30000 | 333445555 | 5 |
| Franklin | T | Wong | 333445555 | 1955-12-08 | 638 Voss, Houston,TX | M | 40000 | 888665555 | 5 |
| Ramesh | K | Narayan | 666884444 | 1962-09-15 | 975 Fire Oak, Humble,TX | M | 38000 | 333445555 | 5 |
| Joyce | A | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX | F | 25000 | 333445555 | 5 |

R

| First_name | Last_name | Salary |
|---|---|---|
| John | Smith | 30000 |
| Franklin | Wong | 40000 |
| Ramesh | Narayan | 38000 |
| Joyce | English | 25000 |

**Figure 2.10 Results of a sequence of operations. (a)** $\pi_{Fname,\ Lname,\ Salary}(\sigma_{Dno=5}(EMPLOYEE))$. ***(b)*** Using intermediate relations and renaming of attributes.

It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression. We can also use this technique to **rename** the attributes in the intermediate and result relations. This can be useful in connection with more complex operations such as UNION and JOIN, as we shall see. To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$$\text{TEMP} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$
$$R(\text{First\_name, Last\_name, Salary}) \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{TEMP})$$

These two operations are illustrated in Figure 2.10(b).

We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator. The general RENAME operation when applied to a relation *R* of degree *n* is denoted by any of the following three forms:

$$\rho_{S(B1,\, B2,\, \ldots,\, Bn)}(R) \quad \text{or} \quad \rho_S(R) \quad \text{or} \quad \rho_{(B1,\, B2,\, \ldots,\, Bn)}(R)$$

where the symbol $\rho$ (rho) is used to denote the RENAME operator, $S$ is the new relation name, and $B_1, B_2, \ldots, B_n$ are the new attribute names. The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of $R$ are $(A_1, A_2, \ldots, A_n)$ in that order, then each $A_i$ is renamed as $B_i$.

In SQL, a single query typically represents a complex relational algebra expression. Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

```
SELECT   E.Fname AS First_name, E.Lname AS Last_name, E.Salary AS Salary
FROM     EMPLOYEE AS E
WHERE    E.Dno=5,
```

## 2.5 Relational Algebra Operations from Set Theory

### 2.5.1 The UNION, INTERSECTION, and MINUS Operations

The next group of relational algebra operations is the standard mathematical operations on sets. For example, to retrieve the Social Security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:

$$\boxed{\begin{array}{l} \text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE}) \\ \text{RESULT1} \leftarrow \pi_{\text{Ssn}}(\text{DEP5\_EMPS}) \\ \text{RESULT2(Ssn)} \leftarrow \pi_{\text{Super\_ssn}}(\text{DEP5\_EMPS}) \\ \text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2} \end{array}}$$

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise an employee who works in department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 2.11), while eliminating any duplicates. Thus, the Ssn value '222445555' appears only once in the result.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations $R$ and $S$ as follows:

- UNION: The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in $R$ or in $S$ or in both $R$ and $S$. Duplicate tuples are eliminated.
- INTERSECTION: The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both $R$ and $S$.
- SET DIFFERENCE (or MINUS): The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in $R$ but not in $S$.

We will adopt the convention that the resulting relation has the same attribute names as the *first* relation *R*. It is always possible to rename the attributes in the result using the rename operator.

**RESULT1**

| Ssn |
|---|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |

**RESULT2**

| Ssn |
|---|
| 333445555 |
| 888665555 |

**RESULT**

| Ssn |
|---|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |
| 888665555 |

**Figure 2.11** Result of the UNION operation RESULT ← RESULT1∪RESULT2.

Figure 2.11 illustrates the three operations such as UNION, INTERSECTION and SET DIFFERENCE between the relations STUDENT and INSTRUCTOR.

The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

(a) **STUDENT**

| Fn | Ln |
|---|---|
| Susan | Yao |
| Ramesh | Shah |
| Johnny | Kohler |
| Barbara | Jones |
| Amy | Ford |
| Jimmy | Wang |
| Ernest | Gilbert |

**INSTRUCTOR**

| Fname | Lname |
|---|---|
| John | Smith |
| Ricardo | Browne |
| Susan | Yao |
| Francis | Johnson |
| Ramesh | Shah |

(b)

| Fn | Ln |
|---|---|
| Susan | Yao |
| Ramesh | Shah |
| Johnny | Kohler |
| Barbara | Jones |
| Amy | Ford |
| Jimmy | Wang |
| Ernest | Gilbert |
| John | Smith |
| Ricardo | Browne |
| Francis | Johnson |

(c)

| Fn | Ln |
|---|---|
| Susan | Yao |
| Ramesh | Shah |

(d)

| Fn | Ln |
|---|---|
| Johnny | Kohler |
| Barbara | Jones |
| Amy | Ford |
| Jimmy | Wang |
| Ernest | Gilbert |

(e)

| Fname | Lname |
|---|---|
| John | Smith |
| Ricardo | Browne |
| Francis | Johnson |

**Figure 2.11** The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT

Notice that both UNION and INTERSECTION are *commutative operations*; that is,

$$R \cup S = S \cup R \quad \text{and} \quad R \cap S = S \cap R$$

Both UNION and INTERSECTION can be treated as $n$-ary operations applicable to any number of relations because both are also *associative operations;* that is,

$$R \cup (S \cup T) = (R \cup S) \cup T \quad \text{and} \quad (R \cap S) \cap T = R \cap (S \cap T)$$

The MINUS operation is *not commutative;* that is, in general,

$$R - S \neq S - R$$

Note that INTERSECTION can be expressed in terms of union and set difference as follows:

$R \cap S = ((R \cup S) - (R - S)) - (S - R)$

### 2.5.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

Next, we discuss the **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT** or **CROSS JOIN**—which is denoted by ×. This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible. In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of $R(A1, A2, ..., An) \times S(B1, B2, ..., Bm)$ is a relation $Q$ with degree $n + m$ attributes $Q(A1, A2, ..., An, B1, B2, ..., Bm)$, in that order. The resulting relation $Q$ has one tuple for each combination of tuples—one from $R$ and one from $S$. Hence, if $R$ has $nR$ tuples (denoted as $|R| = nR$), and $S$ has $nS$ tuples, then $R \times S$ will have $nR * nS$ tuples.

The *n*-ary CARTESIAN PRODUCT operation is an extension of the above concept, which produces new tuples by concatenating all possible combinations of tuples from *n* underlying relations. In general, the CARTESIAN PRODUCT operation applied by itself is generally meaningless. It is mostly useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

$$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex='F'}}(\text{EMPLOYEE})$$
$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$$
$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$$

The resulting relations from this sequence of operations are shown in Figure 2.12. The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can SELECT *related tuples only* from the two relations by specifying an appropriate selection condition after the Cartesian product, as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine *related tuples* from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation.

**FEMALE_EMPS**

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|---|---|---|---|---|---|---|---|---|---|
| Alicia | J | Zelaya | 999887777 | 1968-07-19 | 3321Castle, Spring, TX | F | 25000 | 987654321 | 4 |
| Jennifer | S | Wallace | 987654321 | 1941-06-20 | 291Berry, Bellaire, TX | F | 43000 | 888665555 | 4 |
| Joyce | A | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX | F | 25000 | 333445555 | 5 |

**EMPNAMES**

| Fname | Lname | Ssn |
|---|---|---|
| Alicia | Zelaya | 999887777 |
| Jennifer | Wallace | 987654321 |
| Joyce | English | 453453453 |

**EMP_DEPENDENTS**

| Fname | Lname | Ssn | Essn | Dependent_name | Sex | Bdate | ... |
|---|---|---|---|---|---|---|---|
| Alicia | Zelaya | 999887777 | 333445555 | Alice | F | 1986-04-05 | ... |
| Alicia | Zelaya | 999887777 | 333445555 | Theodore | M | 1983-10-25 | ... |
| Alicia | Zelaya | 999887777 | 333445555 | Joy | F | 1958-05-03 | ... |
| Alicia | Zelaya | 999887777 | 987654321 | Abner | M | 1942-02-28 | ... |
| Alicia | Zelaya | 999887777 | 123456789 | Michael | M | 1988-01-04 | ... |
| Alicia | Zelaya | 999887777 | 123456789 | Alice | F | 1988-12-30 | ... |
| Alicia | Zelaya | 999887777 | 123456789 | Elizabeth | F | 1967-05-05 | ... |
| Jennifer | Wallace | 987654321 | 333445555 | Alice | F | 1986-04-05 | ... |
| Jennifer | Wallace | 987654321 | 333445555 | Theodore | M | 1983-10-25 | ... |
| Jennifer | Wallace | 987654321 | 333445555 | Joy | F | 1958-05-03 | ... |
| Jennifer | Wallace | 987654321 | 987654321 | Abner | M | 1942-02-28 | ... |
| Jennifer | Wallace | 987654321 | 123456789 | Michael | M | 1988-01-04 | ... |
| Jennifer | Wallace | 987654321 | 123456789 | Alice | F | 1988-12-30 | ... |
| Jennifer | Wallace | 987654321 | 123456789 | Elizabeth | F | 1967-05-05 | ... |
| Joyce | English | 453453453 | 333445555 | Alice | F | 1986-04-05 | ... |
| Joyce | English | 453453453 | 333445555 | Theodore | M | 1983-10-25 | ... |
| Joyce | English | 453453453 | 333445555 | Joy | F | 1958-05-03 | ... |
| Joyce | English | 453453453 | 987654321 | Abner | M | 1942-02-28 | ... |
| Joyce | English | 453453453 | 123456789 | Michael | M | 1988-01-04 | ... |
| Joyce | English | 453453453 | 123456789 | Alice | F | 1988-12-30 | ... |
| Joyce | English | 453453453 | 123456789 | Elizabeth | F | 1967-05-05 | ... |

**ACTUAL_DEPENDENTS**

| Fname | Lname | Ssn | Essn | Dependent_name | Sex | Bdate | ... |
|---|---|---|---|---|---|---|---|
| Jennifer | Wallace | 987654321 | 987654321 | Abner | M | 1942-02-28 | ... |

**RESULT**

| Fname | Lname | Dependent_name |
|---|---|---|
| Jennifer | Wallace | Abner |

**Figure 2.12** The Cartesian product (Cross Product) operation.

## 2.6 Binary Relational Operations: JOIN and DIVISION

### 2.6.1 The JOIN Operation

The **JOIN** operation, denoted by $\bowtie$, is used to combine *related tuples* from two relations into single "longer" tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations. To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple.We do this by using the JOIN operation and then projecting the result over the necessary attributes, as follows:

$$\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr\_ssn}=\text{Ssn}} \text{EMPLOYEE}$$
$$\text{RESULT} \leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT\_MGR})$$

The first operation is illustrated in Figure 2.13. Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. This referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE. The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. However, JOIN is very important because it is used very frequently when specifying database queries. Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP\_DEPENDENTS})$$

**DEPT_MGR**

| Dname | Dnumber | Mgr_ssn | . . . | Fname | Minit | Lname | Ssn | . . . |
|---|---|---|---|---|---|---|---|---|
| Research | 5 | 333445555 | . . . | Franklin | T | Wong | 333445555 | . . . |
| Administration | 4 | 987654321 | . . . | Jennifer | S | Wallace | 987654321 | . . . |
| Headquarters | 1 | 888665555 | . . . | James | E | Borg | 888665555 | . . . |

**Figure 2.13** Result of the JOIN operation $\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr\_ssn}=\text{Ssn}} \text{EMPLOYEE}$.

These two operations can be replaced with a single JOIN operation as follows:

$$\text{ACTUAL\_DEPENDENTS} \leftarrow \text{EMPNAMES} \bowtie_{\text{Ssn}=\text{Essn}} \text{DEPENDENT}$$

The general form of a JOIN operation on two relations[5] $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is

$$R \bowtie_{\text{<join condition>}} S$$

A general join condition is of the form

<condition> **AND** <condition> **AND**...**AND** <condition>

where each <condition> is of the form *Ai* θ*Bj*, *Ai* is an attribute of *R*, *Bj* is an attribute of *S*, *Ai* and *Bj* have the same domain, and θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠}. A JOIN operation with such a general join condition is called a **THETA JOIN**. Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result. In that sense, the JOIN operation does *not* necessarily preserve all of the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result.

### 2.6.2 Variations of JOIN: The EQUIJOIN and NATURAL JOIN

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN**. Both previous examples were EQUIJOINs. Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple. For example, in Figure 2.12, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result.

Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN**—denoted by *—was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition. The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first.

Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project. In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

$$\text{PROJ\_DEPT} \leftarrow \text{PROJECT} * \rho_{(\text{Dname, Dnum, Mgr\_ssn, Mgr\_start\_date})}(\text{DEPARTMENT})$$

The same query can be done in two steps by creating an intermediate table DEPT as follows:

$$\text{DEPT} \leftarrow \rho_{(\text{Dname, Dnum, Mgr\_ssn, Mgr\_start\_date})}(\text{DEPARTMENT})$$
$$\text{PROJ\_DEPT} \leftarrow \text{PROJECT} * \text{DEPT}$$

The attribute Dnum is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is illustrated in Figure 2.14. In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the DEPARTMENT tuple for the department that controls the project, but *only one join attribute value* is kept. If the attributes on which the natural join is specified already *have the same names in both relations*, renaming is unnecessary. For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write

DEPT_LOCS <- DEPARTMENT * DEPT_LOCATIONS

The resulting relation is shown in Figure 2.14(b).

(a)

PROJ_DEPT

| Pname | Pnumber | Plocation | Dnum | Dname | Mgr_ssn | Mgr_start_date |
|---|---|---|---|---|---|---|
| ProductX | 1 | Bellaire | 5 | Research | 333445555 | 1988-05-22 |
| ProductY | 2 | Sugarland | 5 | Research | 333445555 | 1988-05-22 |
| ProductZ | 3 | Houston | 5 | Research | 333445555 | 1988-05-22 |
| Computerization | 10 | Stafford | 4 | Administration | 987654321 | 1995-01-01 |
| Reorganization | 20 | Houston | 1 | Headquarters | 888665555 | 1981-06-19 |
| Newbenefits | 30 | Stafford | 4 | Administration | 987654321 | 1995-01-01 |

(b)

DEPT_LOCS

| Dname | Dnumber | Mgr_ssn | Mgr_start_date | Location |
|---|---|---|---|---|
| Headquarters | 1 | 888665555 | 1981-06-19 | Houston |
| Administration | 4 | 987654321 | 1995-01-01 | Stafford |
| Research | 5 | 333445555 | 1988-05-22 | Bellaire |
| Research | 5 | 333445555 | 1988-05-22 | Sugarland |
| Research | 5 | 333445555 | 1988-05-22 | Houston |

**Figure 2.14** Results of two NATURAL JOIN operations. (a) PROJ_DEPT <- PROJECT * DEPT. (b) DEPT_LOCS <- DEPARTMENT * DEPT_LOCATIONS.

### 2.6.3 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations {σ, π,∪,ρ,–, ×} (give examples if asked in detail) is a **complete** set; that is, any of the other original relational algebra operations can be expressed as a *sequence of operations from this set*.

### 2.6.4 The DIVISION Operation

The DIVISION operation, denoted by ÷, is useful for a special kind of query that sometimes occurs in database applications. An example is *Retrieve the names of employees who work on **all** the projects that 'John Smith' works on*. To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

$$SMITH \leftarrow \sigma_{Fname='John' \text{ AND } Lname='Smith'}(EMPLOYEE)$$
$$SMITH\_PNOS \leftarrow \pi_{Pno}(WORKS\_ON \bowtie_{Essn=Ssn} SMITH)$$

Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

$$SSN\_PNOS \leftarrow \pi_{Essn,\, Pno}(WORKS\_ON)$$

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' Social Security numbers:

$$SSNS(Ssn) \leftarrow SSN\_PNOS \div SMITH\_PNOS$$
$$RESULT \leftarrow \pi_{Fname,\, Lname}(SSNS * EMPLOYEE)$$

The preceding operations are shown in Figure 2.15.

**(a)**

**SSN_PNOS**

| Essn | Pno |
|---|---|
| 123456789 | 1 |
| 123456789 | 2 |
| 666884444 | 3 |
| 453453453 | 1 |
| 453453453 | 2 |
| 333445555 | 2 |
| 333445555 | 3 |
| 333445555 | 10 |
| 333445555 | 20 |
| 999887777 | 30 |
| 999887777 | 10 |
| 987987987 | 10 |
| 987987987 | 30 |
| 987654321 | 30 |
| 987654321 | 20 |
| 888665555 | 20 |

**SMITH_PNOS**

| Pno |
|---|
| 1 |
| 2 |

**SSNS**

| Ssn |
|---|
| 123456789 |
| 453453453 |

**(b)**

**R**

| A | B |
|---|---|
| a1 | b1 |
| a2 | b1 |
| a3 | b1 |
| a4 | b1 |
| a1 | b2 |
| a3 | b2 |
| a2 | b3 |
| a3 | b3 |
| a4 | b3 |
| a1 | b4 |
| a2 | b4 |
| a3 | b4 |

**S**

| A |
|---|
| a1 |
| a2 |
| a3 |

**T**

| B |
|---|
| b1 |
| b4 |

**Figure 2.15** The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) T <- R ÷ S

### 2.6.5 Notation for Query Trees

A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Figure 2.16 shows a query tree for Query: *For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.* This query corresponds to the following relational algebra expression:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})) \bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$

In Figure 2.16, the three leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE.

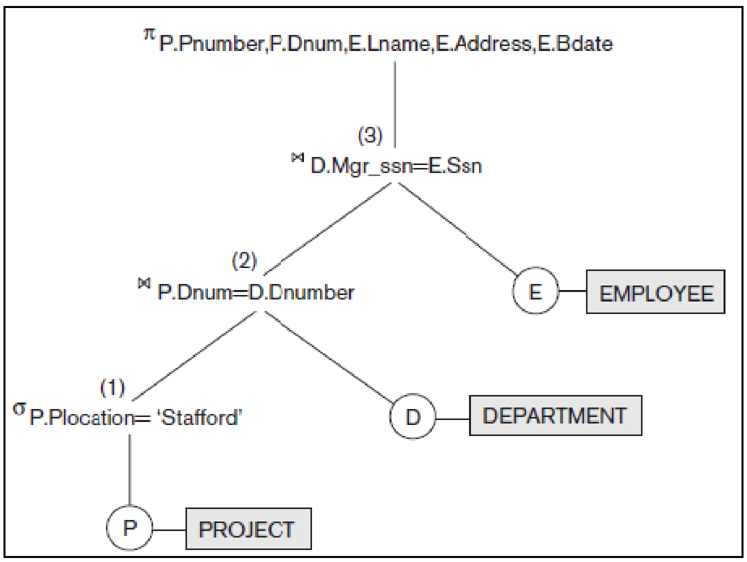


**Figure 2.16** Query tree corresponding to the relational algebra expression for Q2.

## 2.7 Additional Relational Operations

Some common database requests—which are needed in commercial applications for RDBMSs—cannot be performed with the original relational algebra operations. In this section we define additional operations to express these requests. These operations enhance the expressive power of the original relational algebra.

### 2.7.1 Generalized Projection

The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list. The generalized form can be expressed as:

$\pi_{F1, F2, ..., Fn}(R)$

where *F*1, *F*2, ..., *Fn* are functions over the attributes in relation *R* and may involve arithmetic operations and constant values. This operation is helpful when developing reports where computed values have to be produced in the columns of a query result. As an example, consider the relation

As an example, consider the relation

EMPLOYEE (Ssn, Salary, Deduction, Years_service)

A report may be required to show

Net Salary = Salary – Deduction,
Bonus = 2000 * Years_service, and
Tax = 0.25 * Salary.

Then a generalized projection combined with renaming may be used as follows:

$$\text{REPORT} \leftarrow \rho_{(\text{Ssn, Net\_salary, Bonus, Tax})}(\pi_{\text{Ssn, Salary} - \text{Deduction}, 2000 * \text{Years\_service}, 0.25 * \text{Salary}}(\text{EMPLOYEE})).$$

### 2.7.2 Aggregate Functions and Grouping

Another type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database. Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples. These functions are used in simple statistical queries that summarize information from the database tuples. Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.

An example would be to group EMPLOYEE tuples by Dno, so that each group includes the tuples for employees working in the same department. We can then list each Dno value along with, say, the average salary of employees within the department, or the number of employees who work in the department.We can define an AGGREGATE FUNCTION operation, using the symbol $\Im$(pronounced *script F*) , to specify these types of requests as follows:

$$_{\text{<grouping attributes>}} \Im_{\text{<function list>}} (R)$$

where <grouping attributes> is a list of attributes of the relation specified in *R*, and <function list> is a list of (<function> <attribute>) pairs. In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM,COUNT—and <attribute> is an attribute of the relation specified by *R*. The resulting relation has the grouping attributes plus one attribute for each element in the function list. Refer Figure 2.17 for aggregate function operations.

a. $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}(_{\text{Dno}}\ \Im\ _{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE}))$.
b. $_{\text{Dno}}\ \Im\ _{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.
c. $\Im\ _{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

(a) R

| Dno | No_of_employees | Average_sal |
|---|---|---|
| 5 | 4 | 33250 |
| 4 | 3 | 31000 |
| 1 | 1 | 55000 |

(b)

| Dno | Count_ssn | Average_salary |
|---|---|---|
| 5 | 4 | 33250 |
| 4 | 3 | 31000 |
| 1 | 1 | 55000 |

(c)

| Count_ssn | Average_salary |
|---|---|
| 8 | 35125 |

**Figure 2.17 Aggregate function operations.**

### 2.7.3 Recursive Closure Operations

Another type of operation that, in general, cannot be specified in the basic original relational algebra is **recursive closure.** This operation is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor.

example of a recursive operation is to retrieve all supervisees of an employee $e$ at all levels—that is, all employees $e'$ directly supervised by $e$, all employees $e'\Im$ directly supervised by each employee $e'$, all employees $e'''$ directly supervised by each employee $e''$, and so on.

$$\text{BORG\_SSN} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Fname='James' AND Lname='Borg'}}(\text{EMPLOYEE}))$$
$$\text{SUPERVISION(Ssn1, Ssn2)} \leftarrow \pi_{\text{Ssn,Super\_ssn}}(\text{EMPLOYEE})$$
$$\text{RESULT1(Ssn)} \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2=Ssn}} \text{BORG\_SSN})$$

To retrieve all employees supervised by Borg at level 2—that is, all employees $e''$ supervised by some employee $e'$ who is directly supervised by Borg—we can apply another **JOIN** to the result of the first query, as follows:

$$\text{RESULT2(Ssn)} \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2=Ssn}} \text{RESULT1})$$

To get both sets of employees supervised at levels 1 and 2 by 'James Borg', we can apply the UNION operation to the two results, as follows:

$$\text{RESULT} \leftarrow \text{RESULT2} \cup \text{RESULT1}$$

The results of these queries are illustrated in Figure 2.18. Although it is possible to retrieve employees at each level and then take their UNION, we cannot, in general, specify a query such as "retrieve the supervisees of 'James Borg' at all levels" without utilizing a looping mechanism unless we know the maximum number of levels.

**SUPERVISION**

(Borg's Ssn is 888665555)

| (Ssn) Ssn1 | (Super_ssn) Ssn2 |
|---|---|
| 123456789 | 333445555 |
| 333445555 | 888665555 |
| 999887777 | 987654321 |
| 987654321 | 888665555 |
| 666884444 | 333445555 |
| 453453453 | 333445555 |
| 987987987 | 987654321 |
| 888665555 | null |

**RESULT1**

| Ssn |
|---|
| 333445555 |
| 987654321 |

(Supervised by Borg)

**RESULT2**

| Ssn |
|---|
| 123456789 |
| 999887777 |
| 666884444 |
| 453453453 |
| 987987987 |

(Supervised by Borg's subordinates)

**RESULT**

| Ssn |
|---|
| 123456789 |
| 999887777 |
| 666884444 |
| 453453453 |
| 987987987 |
| 333445555 |
| 987654321 |

(RESULT1 ∪ RESULT2)

**Figure 2.18** A two-level recursive query.

### 2.7.4 OUTER JOIN Operations

The JOIN operations described earlier match tuples that satisfy the join condition. For example, for a NATURAL JOIN operation *R* * *S*, only tuples from *R* that have matching tuples in *S*—and vice versa—appear in the result. Hence, tuples without a *matching* (or *related*) tuple are eliminated from the JOIN result. Tuples with NULL values in the join attributes are also eliminated. This type of join, where tuples with no match are eliminated, is known as an **inner join**. This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.

A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in *R*, or all those in *S*, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values.

For example, suppose that we want a list of all employee names as well as the name of the departments they manage *if they happen to manage a department*; if they do not manage one, we can indicate it with a NULL value. We can apply an operation **LEFT OUTER JOIN**, denoted by $⟕$, to retrieve the result as follows:

$$\text{TEMP} \leftarrow (\text{EMPLOYEE} ⟕_{\text{Ssn}=\text{Mgr\_ssn}} \text{DEPARTMENT})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Minit, Lname, Dname}}(\text{TEMP})$$

The LEFT OUTER JOIN operation keeps every tuple in the *first*, or *left*, relation *R* in *R* ⟕ *S*; if no matching tuple is found in *S*, then the attributes of *S* in the join result are filled or *padded* with NULL values. The result of these operations is shown in Figure 2.19. A similar operation, **RIGHT OUTER JOIN**, denoted by ⟖ , keeps every tuple in the *second*, or right, relation *S* in the result of *R* ⟖ *S*. A third operation, **FULL OUTER JOIN**, denoted by ⟗ , keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed.

**RESULT**

| Fname | Minit | Lname | Dname |
|---|---|---|---|
| John | B | Smith | NULL |
| Franklin | T | Wong | Research |
| Alicia | J | Zelaya | NULL |
| Jennifer | S | Wallace | Administration |
| Ramesh | K | Narayan | NULL |
| Joyce | A | English | NULL |
| Ahmad | V | Jabbar | NULL |
| James | E | Borg | Headquarters |

**Figure 2.19** The result of a LEFT OUTER JOIN operation.

### 2.7.5 The OUTER UNION Operation

The **OUTER UNION** operation was developed to take the union of tuples from two relations that have some common attributes, but are *not union (type) compatible*. This operation will take the UNION of tuples in two relations *R*(*X*, *Y*) and *S*(*X*, *Z*) that are **partially compatible**, meaning that only some of their attributes, say *X*, are union compatible. The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation *T*(*X*, *Y*, *Z*). It is therefore the same as a FULL OUTER JOIN on the common attributes.

## 2.8 Examples of Queries in Relational Algebra

The following are additional examples to illustrate the use of the relational algebra operations. All examples refer to the database in Figure 2.7. In general, the same query can be stated in numerous ways using the various operations. We will state each query in one way and leave it to the reader to come up with equivalent formulations.

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

$$\text{RESEARCH\_DEPT} \leftarrow \sigma_{\text{Dname}=\text{'Research'}}(\text{DEPARTMENT})$$
$$\text{RESEARCH\_EMPS} \leftarrow (\text{RESEARCH\_DEPT} \bowtie_{\text{Dnumber}=\text{Dno}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Address}}(\text{RESEARCH\_EMPS})$$

As a single in-line expression, this query becomes:

$$\pi_{\text{Fname, Lname, Address}}\,(\sigma_{\text{Dname}=\text{'Research'}}(\text{DEPARTMENT} \bowtie_{\text{Dnumber}=\text{Dno}}(\text{EMPLOYEE}))$$

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

$$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})$$
$$\text{CONTR\_DEPTS} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT})$$
$$\text{PROJ\_DEPT\_MGRS} \leftarrow (\text{CONTR\_DEPTS} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(\text{PROJ\_DEPT\_MGRS})$$

**Query 2:** Find the names of employees who work on *all* the projects controlled by department number 5.

$$\text{DEPT5\_PROJS} \leftarrow \rho_{(\text{Pno})}(\pi_{\text{Pnumber}}(\sigma_{\text{Dnum=5}}(\text{PROJECT})))$$
$$\text{EMP\_PROJ} \leftarrow \rho_{(\text{Ssn, Pno})}(\pi_{\text{Essn, Pno}}(\text{WORKS\_ON}))$$
$$\text{RESULT\_EMP\_SSNS} \leftarrow \text{EMP\_PROJ} \div \text{DEPT5\_PROJS}$$
$$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{RESULT\_EMP\_SSNS} * \text{EMPLOYEE})$$

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

$$\text{SMITHS(Essn)} \leftarrow \pi_{\text{Ssn}}\ (\sigma_{\text{Lname='Smith'}}(\text{EMPLOYEE}))$$
$$\text{SMITH\_WORKER\_PROJS} \leftarrow \pi_{\text{Pno}}(\text{WORKS\_ON} * \text{SMITHS})$$
$$\text{MGRS} \leftarrow \pi_{\text{Lname, Dnumber}}(\text{EMPLOYEE} \bowtie_{\text{Ssn=Mgr\_ssn}} \text{DEPARTMENT})$$
$$\text{SMITH\_MANAGED\_DEPTS(Dnum)} \leftarrow \pi_{\text{Dnumber}}\ (\sigma_{\text{Lname='Smith'}}(\text{MGRS}))$$
$$\text{SMITH\_MGR\_PROJS(Pno)} \leftarrow \pi_{\text{Pnumber}}(\text{SMITH\_MANAGED\_DEPTS} * \text{PROJECT})$$
$$\text{RESULT} \leftarrow (\text{SMITH\_WORKER\_PROJS} \cup \text{SMITH\_MGR\_PROJS})$$

As a single in-line expression, this query becomes:

$$\pi_{\text{Pno}}\ (\text{WORKS\_ON} \bowtie_{\text{Essn=Ssn}}(\pi_{\text{Ssn}}\ (\sigma_{\text{Lname='Smith'}}(\text{EMPLOYEE}))) \cup \pi_{\text{Pno}} \\ ((\pi_{\text{Dnumber}}\ (\sigma_{\text{Lname='Smith'}}(\pi_{\text{Lname, Dnumber}}(\text{EMPLOYEE}))) \bowtie \\ {}_{\text{Ssn=Mgr\_ssn}}\text{DEPARTMENT})) \bowtie_{\text{Dnumber=Dnum}} \text{PROJECT})$$

**Query 5:** List the names of all employees with two or more dependents. Strictly speaking, this query cannot be done in the *basic* (*original*) *relational algebra*.We have to use the AGGREGATE FUNCTION operation with the COUNT aggregate function. We assume that dependents of the *same* employee have *distinct* Dependent_name values.

$$T1(\text{Ssn, No\_of\_dependents}) \leftarrow {}_{\text{Essn}}\ \Im_{\text{COUNT Dependent\_name}}(\text{DEPENDENT})$$
$$T2 \leftarrow \sigma_{\text{No\_of\_dependents}>2}(T1)$$
$$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(T2 * \text{EMPLOYEE})$$

**Query 6:** Retrieve the names of employees who have no dependents. This is an example of the type of query that uses the MINUS (SET DIFFERENCE) operation.

$ALL\_EMPS \leftarrow \pi_{Ssn}(EMPLOYEE)$
$EMPS\_WITH\_DEPS(Ssn) \leftarrow \pi_{Essn}(DEPENDENT)$
$EMPS\_WITHOUT\_DEPS \leftarrow (ALL\_EMPS - EMPS\_WITH\_DEPS)$
$RESULT \leftarrow \pi_{Lname,\ Fname}(EMPS\_WITHOUT\_DEPS * EMPLOYEE)$

As a single in-line expression, this query becomes:

$$\pi_{Lname,\ Fname}((\pi_{Ssn}(EMPLOYEE) - \rho_{Ssn}(\pi_{Essn}(DEPENDENT))) * EMPLOYEE)$$

**Query 7:** List the names of managers who have at least one dependent.
$MGRS(Ssn) \leftarrow \pi_{Mgr\_ssn}(DEPARTMENT)$
$EMPS\_WITH\_DEPS(Ssn) \leftarrow \pi_{Essn}(DEPENDENT)$
$MGRS\_WITH\_DEPS \leftarrow (MGRS \cap EMPS\_WITH\_DEPS)$
$RESULT \leftarrow \pi_{Lname,\ Fname}(MGRS\_WITH\_DEPS * EMPLOYEE)$