chapter **6**

The Relational Algebra and Relational Calculus

In this chapter we discuss the two formal languages for the relational model: the relational algebra and the relational calculus. As we discussed in Chapter 2, a data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining database structure and constraints. The basic set of operations for the relational model is the **relational algebra**. These operations enable a user to specify basic retrieval requests. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

The relational algebra is very important for several reasons. First, it provides a formal foundation for relational model operations. Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in relational database management systems (RDBMSs), as we discuss in Part 4. Third, some of its concepts are incorporated into the SQL standard query language for RDBMSs. Although no commercial RDBMS in use today provides an interface for relational algebra queries, the core operations and functions of any relational system are based on relational algebra operations. We will define these operations in detail in subsequent sections.

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level declarative notation for specifying relational queries. A relational calculus expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in tuple calculus) or over columns of the stored relations (in domain calculus). In a calculus expression, there is *no order of operations* to specify how to retrieve the query result—a calculus expression specifies only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. The relational calculus is important because it has a firm basis in mathematical logic and because the standard query language (SQL) for RDBMSs has some of its foundations in the tuple relational calculus.[1]

The relational algebra is often considered to be an integral part of the relational data model. Its operations can be divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples in the formal relational model. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT. The other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others. First, we describe the SELECT and PROJECT operations in Section 6.1 because they are **unary operations** that operate on single relations. Then we discuss set operations in Section 6.2. In Section 6.3, we discuss JOIN and other complex **binary operations**, which operate on two tables. The COMPANY relational database shown in Figure 5.6 is used for our examples.

Some common database requests cannot be performed with the original relational algebra operations, so additional operations were created to express these requests. These include **aggregate functions**, which are operations that can *summarize* data from the tables, as well as additional types of JOIN and UNION operations. These operations were added to the original relational algebra because of their importance to many database applications, and are described in Section 6.4. We give examples of specifying queries that use relational operations in Section 6.5. Some of these queries are used in subsequent chapters to illustrate various languages.

In Sections 6.6 and 6.7 we describe the other main formal language for relational databases, the **relational calculus**. There are two variations of relational calculus. The tuple relational calculus is described in Section 6.6 and the domain relational calculus is described in Section 6.7. Some of the SQL constructs discussed in Chapter 8 are based on the tuple relational calculus. The relational calculus is a formal language, based on the branch of mathematical logic called predicate calculus.[2] In tuple relational calculus, variables range over tuples, whereas in domain relational calculus, variables range over the domains (values) of attributes. In Appendix D we give an overview of the Query-By-Example (QBE) language, which is a graph-

1. SQL is based on tuple relational calculus, but also incorporates some of the operations from the relational algebra and its extensions, as we shall see in Chapters 8 and 9.

2. In this chapter no familiarity with first-order predicate calculus—which deals with quantified variables and values—is assumed.

ical user-friendly relational language based on domain relational calculus. Section 6.8 summarizes the chapter.

For the reader who is interested in a less detailed introduction to formal relational languages, Sections 6.4, 6.6, and 6.7 may be skipped.

6.1 Unary Relational Operations: SELECT and PROJECT

6.1.1 The SELECT Operation

The SELECT operation is used to select 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 $30,000, 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 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 $\{=, <, \leq, >, \geq, \neq\}$, and <constant value> is a constant value from the attribute domain. Clauses can be arbitrarily connected by the 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

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level declarative notation for specifying relational queries. A relational calculus expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in tuple calculus) or over columns of the stored relations (in domain calculus). In a calculus expression, there is *no order of operations* to specify how to retrieve the query result—a calculus expression specifies only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. The relational calculus is important because it has a firm basis in mathematical logic and because the standard query language (SQL) for RDBMSs has some of its foundations in the tuple relational calculus.[1]

The relational algebra is often considered to be an integral part of the relational data model. Its operations can be divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples in the formal relational model. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT. The other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others. First, we describe the SELECT and PROJECT operations in Section 6.1 because they are **unary operations** that operate on single relations. Then we discuss set operations in Section 6.2. In Section 6.3, we discuss JOIN and other complex **binary operations**, which operate on two tables. The COMPANY relational database shown in Figure 5.6 is used for our examples.

Some common database requests cannot be performed with the original relational algebra operations, so additional operations were created to express these requests. These include **aggregate functions**, which are operations that can *summarize* data from the tables, as well as additional types of JOIN and UNION operations. These operations were added to the original relational algebra because of their importance to many database applications, and are described in Section 6.4. We give examples of specifying queries that use relational operations in Section 6.5. Some of these queries are used in subsequent chapters to illustrate various languages.

In Sections 6.6 and 6.7 we describe the other main formal language for relational databases, the **relational calculus**. There are two variations of relational calculus. The tuple relational calculus is described in Section 6.6 and the domain relational calculus is described in Section 6.7. Some of the SQL constructs discussed in Chapter 8 are based on the tuple relational calculus. The relational calculus is a formal language, based on the branch of mathematical logic called predicate calculus.[2] In tuple relational calculus, variables range over tuples, whereas in domain relational calculus, variables range over the domains (values) of attributes. In Appendix D we give an overview of the Query-By-Example (QBE) language, which is a graph-

1. SQL is based on tuple relational calculus, but also incorporates some of the operations from the relational algebra and its extensions, as we shall see in Chapters 8 and 9.

2. In this chapter no familiarity with first-order predicate calculus—which deals with quantified variables and values—is assumed.

ical user-friendly relational language based on domain relational calculus. Section 6.8 summarizes the chapter.

For the reader who is interested in a less detailed introduction to formal relational languages, Sections 6.4, 6.6, and 6.7 may be skipped.

6.1 Unary Relational Operations: SELECT and PROJECT

6.1.1 The SELECT Operation

The SELECT operation is used to select 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 $30,000, 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 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 $\{=, <, \leq, >, \geq, \neq\}$, and <constant value> is a constant value from the attribute domain. Clauses can be arbitrarily connected by the 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 $30,000, 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 6.1(a).

Notice that the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$ apply to attributes whose domains are *ordered values*, such as numeric or date domains. Domains of strings of characters are considered ordered based on the collating sequence of the characters. If the domain of an attribute is a set of *unordered values*, then only the comparison operators in the set $\{=, \neq\}$ can be used. 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.

In general, the result of a SELECT operation can be determined as follows. The <selection condition> is applied independently to each tuple t in R. This is done by substituting each occurrence of an attribute A_i in the selection condition with its value in the tuple $t[A_i]$. If the condition evaluates to TRUE, then tuple t is **selected.** All the selected tuples appear in the result of the SELECT operation. 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.

Figure 6.1
Results of SELECT and PROJECT operations. (a) $\sigma_{(\text{Dno}=4 \textbf{ AND } \text{Salary}>25000) \text{ OR } (\text{Dno}=5 \textbf{ AND } \text{Salary}>30000)}(\text{EMPLOYEE})$. (b) $\pi_{\text{Lname, Fname, Salary}}(\text{EMPLOYEE})$. (c) $\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$.

(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

- (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 **degree** of the relation resulting from a SELECT operation—its number of attributes—is the same as the degree of *R*. The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in *R*. That is, $|\sigma_c(R)| \leq |R|$ for any condition *C*. 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** of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is,

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

6.1.2 The PROJECT Operation

If we think of a relation as a table, the SELECT operation selects 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. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. 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 6.1(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 list 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 tuples, and hence a valid relation. This is known as **duplicate elimination.** For example, consider the following PROJECT operation:

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

The result is shown in Figure 6.1(c). Notice that the tuple <'F', 25000> appears only once in Figure 6.1(c), even though this combination of values appears twice in the EMPLOYEE relation. Duplicate elimination involves sorting to detect duplicates and hence adds more processing. If duplicates are not eliminated, the result would be a **multiset** or **bag** of tuples rather than a set. This was not permitted in the formal relational model, but is allowed in practice. In Chapter 8 we will see that the user can control whether duplicates should be eliminated or not.

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in *R*. If the projection list is a superkey of *R*—that is, it includes some key of *R*—the resulting relation has the *same number* of tuples as *R*. Moreover,

$$\pi_{<\text{list1}>}\ (\pi_{<\text{list2}>}(R)) = \pi_{<\text{list1}>}(R)$$

as long as <list2> contains the attributes in <list1>; otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity *does not* hold on PROJECT.

6.1.3 Sequences of Operations and the RENAME Operation

The relations shown in Figure 6.1 do not have any names. In general, we may want 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 as follows:

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

Figure 6.2(a) shows the result of this relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation:

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

It is often 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

(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 6.2
Results of a sequence of operations.
(a) $\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$.
(b) Using intermediate relations and renaming of attributes.

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 6.2(b).

If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

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:

$$\rho_{S(B_1, B_2, \ldots, B_n)}(R) \quad \text{or} \quad \rho_S(R) \quad \text{or} \quad \rho_{(B_1, B_2, \ldots, B_n)}(R)$$

where the symbol ρ (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.

6.2 Relational Algebra Operations from Set Theory

6.2.1 The UNION, INTERSECTION, and MINUS Operations

The next group of relational algebra operations are 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:[3]

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

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 6.3). Thus, the Ssn value '333445555' appears only once in the result.

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS**). These are **binary** operations; that is, each is applied to two sets (of tuples). When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called *union compatibility*. Two relations $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_n)$ are said to be **union compatible** if they have the same degree n

Figure 6.3
Result of the UNION operation RESULT ← RESULT1 ∪ RESULT2.

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

3. As a single relational algebra expression, this becomes
$\text{Result} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE})) \cup \pi_{\text{Super\_ssn}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$

and if $\text{dom}(A_i) = \text{dom}(B_i)$ for $1 \le i \le n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

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.

Figure 6.4 illustrates the three operations. The relations STUDENT and INSTRUCTOR in Figure 6.4(a) are union compatible and their tuples represent the names of students and instructors, respectively. The result of the UNION operation in Figure 6.4(b) shows the names of all students and instructors. Note that duplicate tuples appear only once in the result. The result of the INTERSECTION operation (Figure 6.4(c)) includes only those who are both students and instructors.

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 *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,

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

Figure 6.4(d) shows the names of students who are not instructors, and Figure 6.4(e) shows the names of instructors who are not students.

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

$$\mathrm{R} \cap \mathrm{S} = \mathrm{R} \cup \mathrm{S} - (\mathrm{R} - \mathrm{S}) - (\mathrm{S} - \mathrm{R})$$

6.2.2 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 $\times$. This is also a binary set opera-

(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 6.4
The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

tion, 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 $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, 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 n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ 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. The operation applied by itself is generally meaningless. It is 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:

FEMALE_EMPS ← $\sigma_{Sex='F'}$(EMPLOYEE)
EMPNAMES ← $\pi_{Fname,\ Lname,\ Ssn}$(FEMALE_EMPS)
EMP_DEPENDENTS ← EMPNAMES × DEPENDENT
ACTUAL_DEPENDENTS ← $\sigma_{Ssn=Essn}$(EMP_DEPENDENTS)
RESULT ← $\pi_{Fname,\ Lname,\ Dependent\_name}$(ACTUAL_DEPENDENTS)

The resulting relations from this sequence of operations are shown in Figure 6.5. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES from Figure 6.5 with DEPENDENT from Figure 5.6. In EMP_DEPENDENTS, every tuple from EMPNAMES is combined with every tuple from DEPENDENT, giving a result that is not very meaningful. We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose Essn values match the Ssn value of the EMPLOYEE tuple. The ACTUAL_DEPENDENTS relation accomplishes this. The EMP_DEPENDENTS relation is a good example of the case where relational algebra can be correctly applied to yield results that make no sense at all. Therefore, it is the responsibility of the user to make sure to apply only meaningful operations to relations.

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 as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is used quite commonly to identify and select related tuples from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation. We discuss the JOIN operation next.

6.3 Binary Relational Operations: JOIN and DIVISION

6.3.1 The JOIN Operation

The **JOIN** operation, denoted by ⋈, is used to combine *related tuples* from two relations into single 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:

DEPT_MGR ← DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE
RESULT ← $\pi_{Dname,\ Lname,\ Fname}$(DEPT_MGR)

The first operation is illustrated in Figure 6.6. Note that MGR_SSN is a foreign key and that the referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.

Figure 6.5
The CARTESIAN PRODUCT (CROSS PRODUCT) 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

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 6.6
Result of the JOIN operation
DEPT_MGR ← DEPARTMENT $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE.

The JOIN operation can be stated in terms of a CARTESIAN PRODUCT followed by a SELECT operation. However, JOIN is very important because it is used very frequently when specifying database queries. Consider the example we gave earlier to illustrate CARTESIAN PRODUCT, which included the following sequence of operations:

EMP_DEPENDENTS ← EMPNAMES × DEPENDENT
ACTUAL_DEPENDENTS ← $\sigma_{\text{Ssn=Essn}}$(EMP_DEPENDENTS)

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

ACTUAL_DEPENDENTS ← EMPNAMES $\bowtie_{\text{Ssn=Essn}}$ DEPENDENT

The general form of a JOIN operation on two relations[4] $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is

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

The result of the JOIN is a relation Q with $n + m$ attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—*whenever the combination satisfies the join condition.* This is the main difference between CARTESIAN PRODUCT and JOIN. In JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation Q *as a single combined tuple.*

A general join condition is of the form

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

where each condition is of the form $A_i \; \theta \; B_j$, A_i is an attribute of R, B_j is an attribute of S, A_i and B_j have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$. A JOIN operation with such a general join condition is called a

4. Again, notice that R and S can be any relations that result from general *relational algebra expressions.*

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.

6.3.2 *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 examples we have considered 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 6.6, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR because of the equality join condition specified on these two attributes. 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.[5] 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.

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**. The resulting relation is illustrated in Figure 6.7(a). 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* 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

$$\text{DEPT\_LOCS} \leftarrow \text{DEPARTMENT} * \text{DEPT\_LOCATIONS}$$

The resulting relation is shown in Figure 6.7(b), which combines each department with its locations and has one tuple for each location. In general, NATURAL JOIN is performed by equating *all* attribute pairs that have the same name in the two relations. There can be a list of join attributes from each relation, and each corresponding pair must have the same name.

5. NATURAL JOIN is basically an EQUIJOIN followed by removal of the superfluous attributes.

(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 6.7
Results of two NATURAL JOIN operations.
(a) PROJ_DEPT ← PROJECT * DEPT.
(b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

A more general, *but nonstandard* definition for NATURAL JOIN is

$$Q \leftarrow R *_{(\text{<list1>}),(\text{<list2>})} S$$

In this case, <list1> specifies a list of *i* attributes from *R*, and <list2> specifies a list of *i* attributes from *S*. The lists are used to form equality comparison conditions between pairs of corresponding attributes, and the conditions are then ANDed together. Only the list corresponding to attributes of the first relation *R*—<list1>—is kept in the result *Q*.

Notice that if no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples. In general, if *R* has n_R tuples and *S* has n_S tuples, the result of a JOIN operation $R \bowtie_{\text{<join condition>}} S$ will have between zero and $n_R * n_S$ tuples. The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called **join selectivity**, which is a property of each join condition. If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

As we can see, the JOIN operation is used to combine data from multiple relations so that related information can be presented in a single table. These operations are also known as **inner joins**, to distinguish them from a different join variation called *outer joins* (see Section 6.4.4). Informally, an *inner join* is a type of match and merge operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. An *outer join* is another, more lenient version of the same. Note that sometimes a join may be specified between a relation and itself, as we shall illustrate in Section 6.4.3. The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join:

$$((\text{PROJECT} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT}) \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$$

This links each project to its controlling department, and then relates the department to its manager employee. The net result is a consolidated relation in which each tuple contains this project-department-manager information.

6.3.3 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations $\{\sigma, \pi, \cup, -, \times\}$ 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*. For example, the INTERSECTION operation can be expressed by using UNION and MINUS as follows:

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

Although, strictly speaking, INTERSECTION is not required, it is inconvenient to specify this complex expression every time we wish to specify an intersection. As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

$$R \bowtie_{\text{<condition>}} S \equiv \sigma_{\text{<condition>}}(R \times S)$$

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also *not strictly necessary* for the expressive power of the relational algebra. However, they are important to consider as separate operations because they are convenient to use and are very commonly applied in database applications. One may include the RENAME operation as an essential operation if the need to rename the result of a relational algebra expression is considered as a necessity. Other operations have been included in the relational algebra for convenience rather than necessity. We discuss one of these—the DIVISION operation—in the next section.

6.3.4 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:

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

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:

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

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

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

The previous operations are shown in Figure 6.8(a).

In general, the DIVISION operation is applied to two relations $R(Z) \div S(X)$, where $X \subseteq Z$. Let $Y = Z - X$ (and hence $Z = X \cup Y$); that is, let Y be the set of attributes of R that are not attributes of S. The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S$ for *every* tuple t_S in S. This means that, for a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with every tuple in S. Note that in the formulation of the DIVISION operation, the tuples in the denominator relation restrict the numerator relation by selecting those tuples in the result that match all values present in the denominator. It is not necessary to know what those values are.

Figure 6.8(b) illustrates a DIVISION operation where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$. Notice that the tuples (values) b_1 and b_4 appear in R in combination with all three tuples in S; that is why they appear in the resulting relation T. All other values of B in R do not appear with all the tuples in S and are not selected: b_2 does not appear with a_2, and b_3 does not appear with a_1.

The DIVISION operation can be expressed as a sequence of π, $\times$, and $-$ operations as follows:

$$T1 \leftarrow \pi_Y(R)$$
$$T2 \leftarrow \pi_Y((S \times T1) - R)$$
$$T \leftarrow T1 - T2$$

The DIVISION operation is defined for convenience for dealing with queries that involve *universal quantification* (see Section 6.6.7) or the *all* condition. Most RDBMS implementations with SQL as the primary query language do not directly implement division. SQL has a roundabout way of dealing with the type of query illustrated above (see Section 8.5.4). Table 6.1 lists the various basic relational algebra operations we have discussed.

(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 6.8
The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

6.3.5 Notation for Query Trees

In this section we describe a notation typically used in relational systems to represent queries internally. The notation is called a query tree or sometimes it is known as a query evaluation tree or query execution tree. It includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.

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 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 6.9 shows a query tree for query Q2: *For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birth date.* This query is specified on the

Table 6.1
Operations of Relational Algebra

Operation	Purpose	Notation
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{\text{<selection condition>}}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{\text{<attribute list>}}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\text{<join condition>}} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\text{<join condition>}} R_2$, OR $R_1 \bowtie_{(\text{<join attributes 1>}), (\text{<join attributes 2>})} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{\text{<join condition>}} R_2$, OR $R_1 *_{(\text{<join attributes 1>}), (\text{<join attributes 2>})} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

relational schema of Figure 5.5 and 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 6.9 the three relations PROJECT, DEPARTMENT, and EMPLOYEE are represented by leaf nodes P, D, and E, while the relational algebra operations of the expres-

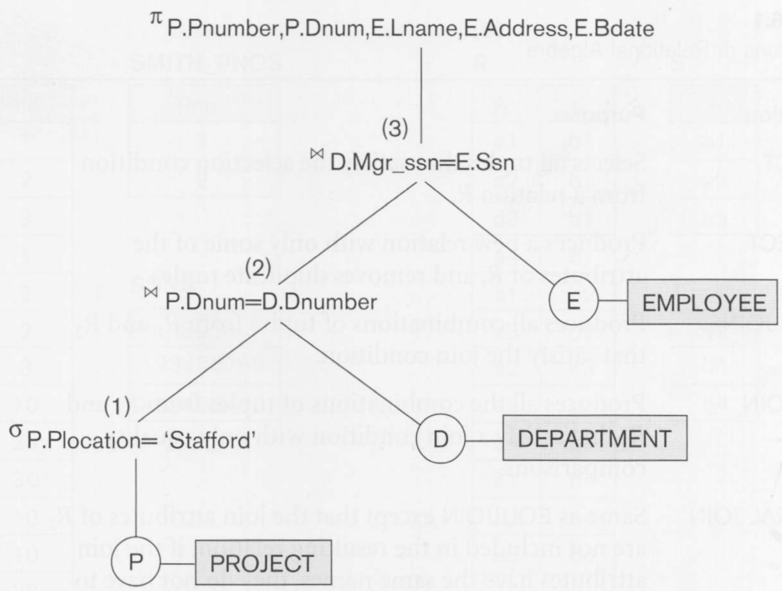


Figure 6.9
Query tree corresponding to the relational algebra expression for Q2.

sion are represented by internal tree nodes. It signifies an order of execution in the following sense. In order to execute Q2, the node marked (1) in Figure 6.9 must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2). Similarly, node (2) must begin to execute and produce results before node (3) can start execution, and so on. In general, a query tree gives a good visual representation and understanding of the query in terms of the relational operations it uses and is recommended as an additional means for expressing queries in relational algebra. We will revisit query trees when we discuss query processing and optimization in Chapter 15.

6.4 Additional Relational Operations

Some common database requests—which are needed in commercial applications for RDBMSs—cannot be performed with the original relational algebra operations described in Sections 6.1 through 6.3. In this section we define additional operations to express these requests. These operations enhance the expressive power of the original relational algebra.

6.4.1 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:

$$\pi_{F_1, F_2, \ldots, F_n}(R)$$

where $F_1, F_2, ..F_n$ are functions over the attributes in relation R and may involve constants. This operation is devised as a helpful operation when developing reports where computed values have to be produced in columns.

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:

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

6.4.2 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.

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group. 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*)[6], 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. For example, to retrieve each department number, the

6. There is no single agreed-upon notation for specifying aggregate functions. In some cases a "script A" is used.

number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write:

$$\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}\left({}_{\text{Dno}}\,\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})\right)$$

The result of this operation on the EMPLOYEE relation of Figure 5.6 is shown in Figure 6.10(a).

In the above example, we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation *R*. If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form <function>_<attribute>.[7] For example, Figure 6.10(b) shows the result of the following operation:

$$_{\text{Dno}}\,\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$$

If no grouping attributes are specified, the functions are applied to *all the tuples* in the relation, so the resulting relation has a *single tuple only*. For example, Figure 6.10(c) shows the result of the following operation:

$$\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$$

It is important to note that, in general, duplicates are *not eliminated* when an aggregate function is applied; this way, the normal interpretation of functions such as SUM and AVERAGE is computed.[8] It is worth emphasizing that the result of apply-

Figure 6.10
The aggregate function operation.
(a) $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}\left({}_{\text{Dno}}\,\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})\right)$.
(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

7. Note that this is an arbitrary notation we are suggesting. There is no standard notation.

8. In SQL, the option of eliminating duplicates before applying the aggregate function is available by including the keyword DISTINCT (see Section 8.4.4).

ing an aggregate function is a relation, not a scalar number—even if it has a single value. This makes the relational algebra a closed system.

6.4.3 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. This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation in Figures 5.5 and 5.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor). An 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″* directly supervised by each employee *e′*; all employees *e‴* directly supervised by each employee *e″*; and so on.

Although it is straightforward in the relational algebra to specify all employees supervised by *e at a specific level*, it is difficult to specify all supervisees at *all* levels. For example, to specify the Ssns of all employees *e′* directly supervised—*at level one*—by the employee *e* whose name is 'James Borg' (see Figure 5.6), we can apply the following operation:

$$\text{BORG\_SSN} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Fname='James' AND Lname='Borg'}}(\text{EMPLOYEE}))$$

$$\text{SUPERVISION(Ssn1, Ssn2)} \leftarrow \pi_{\text{Ssn, Super\_ssn}}(\text{EMPLOYEE})$$

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

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 6.11. 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.[9] An operation called the *transitive closure* of relations has been proposed to compute the recursive relationship as far as the recursion proceeds.

9. The SQL3 standard includes syntax for recursive closure.

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 6.11
A two-level recursive query.

6.4.4 OUTER JOIN Operations

Next, we discuss some extensions to the JOIN operation that are necessary to specify certain types of queries. 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 amounts to loss of information, if the result of JOIN is supposed to be used to generate a report based on all the information in the component relations.

A set of operations, called **outer joins,** can be used when we want 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. The join operations we described earlier in Section 6.3, where only matching tuples are kept in the result, are called **inner joins**.

For example, suppose that we want a list of all employee names and also 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:

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

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 6.12.

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. The three outer join operations are part of the SQL2 standard (see Chapter 8). These operations were provided later as an extension of relational algebra in response to the typical need in business applications to show related information from multiple tables exhaustively. Sometimes a complete reporting of data from multiple tables is required whether or not there are matching values.

6.4.5 The OUTER UNION Operation

The **OUTER UNION** operation was developed to take the union of tuples from two relations if the relations are *not union compatible*. This operation will take the

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 6.12
The result of a LEFT OUTER JOIN operation.

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 $T(X, Y, Z)$.

Two tuples t_1 in R and t_2 in S are said to **match** if $t_1[X]=t_2[X]$, and are considered to represent the same entity or relationship instance. These will be combined (unioned) into a single tuple in T. Tuples in either relation that have no matching tuple in the other relation are padded with NULL values. For example, an OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, Ssn, Department, Advisor) and INSTRUCTOR(Name, Ssn, Department, Rank). Tuples from the two relations are matched based on having the same combination of values of the shared attributes—Name, Ssn, Department. The result relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

STUDENT_OR_INSTRUCTOR(Name, Ssn, Department, Advisor, Rank)

All the tuples from both relations are included in the result, but tuples with the same (Name, Ssn, Department) combination will appear only once in the result. Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute. A tuple that exists in both relations, such as a student who is also an instructor, will have values for all its attributes.[10]

Notice that the same person may still appear twice in the result. For example, we could have a graduate student in the Mathematics department who is an instructor in the Computer Science department. Although the two tuples representing that person in STUDENT and INSTRUCTOR will have the same (Name, Ssn) values, they will not agree on the Department value, and so will not be matched. This is because Department has two separate meanings in STUDENT (the department where the person studies) and INSTRUCTOR (the department where the person is employed as an instructor). If we wanted to union persons based on the same (Name, Ssn) combination only, we should rename the Department attribute in each table to reflect that they have different meanings and designate them as not being part of the union-compatible attributes.

Another capability that exists in most commercial languages (but not in the basic relational algebra) is that of specifying operations on values after they are extracted from the database. For example, arithmetic operations such as +, –, and * can be applied to numeric values that appear in the result of a query, as we discussed in Section 6.4.1.

---

10. Notice that OUTER UNION is equivalent to a FULL OUTER JOIN if the join attributes are *all* the common attributes of the two relations.

6.5 Examples of Queries in Relational Algebra

Following, we give additional examples to illustrate the use of the relational algebra operations. All examples refer to the database of Figure 5.6. 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.

$\text{RESEARCH\_DEPT} \leftarrow \sigma_{\text{Dname='Research'}}(\text{DEPARTMENT})$

$\text{RESEARCH\_EMPS} \leftarrow (\text{RESEARCH\_DEPT} \bowtie_{\text{Dnumber=Dno}} \text{EMPLOYEE})$

$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Address}}(\text{RESEARCH\_EMPS})$

As a single expression, this query becomes

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

This query could be specified in other ways; for example, the order of the JOIN and SELECT operations could be reversed, or the JOIN could be replaced by a NATURAL JOIN after renaming one of the join attributes.

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.

$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})$

$\text{CONTR\_DEPT} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT})$

$\text{PROJ\_DEPT\_MGR} \leftarrow (\text{CONTR\_DEPT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$

$\text{RESULT} \leftarrow \pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(\text{PROJ\_DEPT\_MGR})$

Query 3. Find the names of employees who work on *all* the projects controlled by department number 5.

$\text{DEPT5\_PROJS(Pno)} \leftarrow \pi_{\text{Pnumber}}(\sigma_{\text{Dnum=5}}(\text{PROJECT}))$

$\text{EMP\_PROJ(Ssn, Pno)} \leftarrow \pi_{\text{Essn, Pno}}(\text{WORKS\_ON})$

$\text{RESULT\_EMP\_SSNS} \leftarrow \text{EMP\_PROJ} \div \text{DEPT5\_PROJS}$

$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{RESULT\_EMP\_SSNS} * \text{EMPLOYEE})$

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 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.

$T1(\text{Ssn, No\_of\_dependents}) \leftarrow {}_{\text{Essn}} \Im_{\text{ COUNT Dependent\_name}}(\text{DEPENDENT})$

$T2 \leftarrow \sigma_{\text{No\_of\_dependents} \geq 2}(T1)$

$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(T2 * \text{EMPLOYEE})$

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.

$\text{ALL\_EMPS} \leftarrow \pi_{\text{Ssn}}(\text{EMPLOYEE})$

$\text{EMPS\_WITH\_DEPS(Ssn)} \leftarrow \pi_{\text{Essn}}(\text{DEPENDENT})$

$\text{EMPS\_WITHOUT\_DEPS} \leftarrow (\text{ALL\_EMPS} - \text{EMPS\_WITH\_DEPS})$

$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{EMPS\_WITHOUT\_DEPS} * \text{EMPLOYEE})$

As a single expression, this query becomes

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

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

$\text{MGRS(Ssn)} \leftarrow \pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$

$\text{EMPS\_WITH\_DEPS(Ssn)} \leftarrow \pi_{\text{Essn}}(\text{DEPENDENT})$

$\text{MGRS\_WITH\_DEPS} \leftarrow (\text{MGRS} \cap \text{EMPS\_WITH\_DEPS})$

$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{MGRS\_WITH\_DEPS} * \text{EMPLOYEE})$

As we mentioned earlier, in general, the same query can be specified in many different ways. For example, the operations can often be applied in various orders. In addition, some operations can be used to replace others; for example, the INTERSECTION operation in Q7 can be replaced by a NATURAL JOIN. As an exercise, try to do each of the above example queries using different operations.[11] We showed how to write queries as single relational algebra expressions for queries Q1, Q4, and Q6. Try to write the remaining queries as single expressions. In Chapter 8 and in Sections 6.6 and 6.7, we show how these queries are written in other relational languages.

6.6 The Tuple Relational Calculus

In this and the next section, we introduce another formal query language for the relational model called **relational calculus**. In relational calculus, we write one **declarative** expression to specify a retrieval request; hence, there is no description of how to evaluate a query. A calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **non-procedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request; hence, it can be considered as a **procedural** way of stating a query. It is possible to nest algebra operations to form a single expression; however, a certain order among the operations is always explicitly specified in a relational algebra expression. This order also influences the strategy for evaluating the query. A calculus expression may be written in different ways, but the way it is written has no bearing on how a query should be evaluated.

It has been shown that any retrieval that can be specified in the basic relational algebra can also be specified in relational calculus, and vice versa; in other words, the **expressive power** of the two languages is *identical*. This led to the definition of the concept of a relationally complete language. A relational query language *L* is considered **relationally complete** if we can express in *L* any query that can be expressed in relational calculus. Relational completeness has become an important basis for comparing the expressive power of high-level query languages. However, as we saw in Section 6.4, certain frequently required queries in database applications cannot be expressed in basic relational algebra or calculus. Most relational query languages are relationally complete but have *more expressive power* than relational algebra or relational calculus because of additional operations such as aggregate functions, grouping, and ordering.

In this section and the next, all our examples refer to the database shown in Figures 5.6 and 5.7. We will use the same queries that were used in Section 6.5. Sections 6.6.6, 6.6.7, and 6.6.8 discuss dealing with universal quantifiers and safety of expression issues and may be skipped by students interested in a general introduction to tuple calculus.

6.6.1 Tuple Variables and Range Relations

The tuple relational calculus is based on specifying a number of **tuple variables**. Each tuple variable usually *ranges over* a particular database relation, meaning that

11. When queries are optimized (see Chapter 15), the system will choose a particular sequence of operations that corresponds to an execution strategy that can be executed efficiently.

the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form

$\{t \mid \text{COND}(t)\}$

where t is a tuple variable and COND(t) is a conditional expression involving t. The result of such a query is the set of all tuples t that satisfy COND(t). For example, to find all employees whose salary is above $50,000, we can write the following tuple calculus expression:

{t | EMPLOYEE(t) **AND** t.Salary>50000}

The condition EMPLOYEE(t) specifies that the **range relation** of tuple variable t is EMPLOYEE. Each EMPLOYEE tuple t that satisfies the condition t.Salary>50000 will be retrieved. Notice that t.Salary references attribute Salary of tuple variable t; this notation resembles how attribute names are qualified with relation names or aliases in SQL, as we shall see in Chapter 8. In the notation of Chapter 5, t.Salary is the same as writing t[Salary].

The above query retrieves all attribute values for each selected EMPLOYEE tuple t. To retrieve only *some* of the attributes—say, the first and last names—we write

{t.Fname, t.Lname | EMPLOYEE(t) **AND** t.Salary>50000}

Informally, we need to specify the following information in a tuple calculus expression:

- For each tuple variable t, the **range relation** R of t. This value is specified by a condition of the form $R(t)$.
- A condition to select particular combinations of tuples. As tuple variables range over their respective range relations, the condition is evaluated for every possible combination of tuples to identify the **selected combinations** for which the condition evaluates to TRUE.
- A set of attributes to be retrieved, the **requested attributes**. The values of these attributes are retrieved for each selected combination of tuples.

Before we discuss the formal syntax of tuple relational calculus, consider another query.

Query 0. Retrieve the birth date and address of the employee (or employees) whose name is John B. Smith.

Q0: {t.Bdate, t.Address | EMPLOYEE(t) **AND** t.Fname='John'
AND t.Minit='B' **AND** t.Lname='Smith'}

In tuple relational calculus, we first specify the requested attributes t.Bdate and t.Address for each selected tuple t. Then we specify the condition for selecting a tuple following the bar (|)—namely, that t be a tuple of the EMPLOYEE relation whose Fname, Minit, and Lname attribute values are 'John', 'B', and 'Smith', respectively.

6.6.2 Expressions and Formulas in Tuple Relational Calculus

A general **expression** of the tuple relational calculus is of the form

$$\{t_1.A_j, t_2.A_k, \ldots, t_n.A_m \mid \text{COND}(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$$

where $t_1, t_2, \ldots, t_n, t_{n+1}, \ldots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and COND is a **condition** or **formula**[12] of the tuple relational calculus. A formula is made up of predicate calculus **atoms,** which can be one of the following:

1. An atom of the form $R(t_i)$, where R is a relation name and t_i is a tuple variable. This atom identifies the range of the tuple variable t_i as the relation whose name is R.
2. An atom of the form $t_i.A$ **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, and B is an attribute of the relation on which t_j ranges.
3. An atom of the form $t_i.A$ **op** c or c **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, B is an attribute of the relation on which t_j ranges, and c is a constant value.

Each of the preceding atoms evaluates to either TRUE or FALSE for a specific combination of tuples; this is called the **truth value** of an atom. In general, a tuple variable t ranges over all possible tuples *in the universe.* For atoms of the form $R(t)$, if t is assigned to a tuple that is a *member of the specified relation* R, the atom is TRUE; otherwise, it is FALSE. In atoms of types 2 and 3, if the tuple variables are assigned to tuples such that the values of the specified attributes of the tuples satisfy the condition, then the atom is TRUE.

A **formula** (condition) is made up of one or more atoms connected via the logical operators **AND**, **OR**, and **NOT** and is defined recursively by Rules 1 and 2 as follows:

- *Rule 1*: Every atom is a formula.
- *Rule 2*: If F_1 and F_2 are formulas, then so are (F_1 **AND** F_2), (F_1 **OR** F_2), **NOT** (F_1), and **NOT** (F_2). The truth values of these formulas are derived from their component formulas F_1 and F_2 as follows:
  a. (F_1 **AND** F_2) is TRUE if both F_1 and F_2 are TRUE; otherwise, it is FALSE.
  b. (F_1 **OR** F_2) is FALSE if both F_1 and F_2 are FALSE; otherwise, it is TRUE.
  c. **NOT** (F_1) is TRUE if F_1 is FALSE; it is FALSE if F_1 is TRUE.
  d. **NOT** (F_2) is TRUE if F_2 is FALSE; it is FALSE if F_2 is TRUE.

12. Also called a **well-formed formula**, or **WFF**, in mathematical logic.

6.6.3 The Existential and Universal Quantifiers

In addition, two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** ($\forall$) and the **existential quantifier** ($\exists$). Truth values for formulas with quantifiers are described in Rules 3 and 4 below; first, however, we need to define the concepts of free and bound tuple variables in a formula. Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an $(\exists t)$ or $(\forall t)$ clause; otherwise, it is free. Formally, we define a tuple variable in a formula as **free** or **bound** according to the following rules:

- An occurrence of a tuple variable in a formula F that *is an atom* is free in F.
- An occurrence of a tuple variable t is free or bound in a formula made up of logical connectives—(F_1 **AND** F_2), (F_1 **OR** F_2), **NOT**(F_1), and **NOT**(F_2)—depending on whether it is free or bound in F_1 or F_2 (if it occurs in either). Notice that in a formula of the form $F = (F_1$ **AND** $F_2)$ or $F = (F_1$ **OR** $F_2)$, a tuple variable may be free in F_1 and bound in F_2, or vice versa; in this case, one occurrence of the tuple variable is bound and the other is free in F.
- All *free* occurrences of a tuple variable t in F are **bound** in a formula F' of the form $F' = (\exists\ t)(F)$ or $F' = (\forall\ t)(F)$. The tuple variable is bound to the quantifier specified in F'. For example, consider the following formulas:

  F_1 : d.Dname='Research'

  F_2 : $(\exists t)(d$.Dnumber=t.Dno)

  F_3 : $(\forall d)(d$.Mgr_ssn='333445555')

The tuple variable d is free in both F_1 and F_2, whereas it is bound to the ($\forall$) quantifier in F_3. Variable t is bound to the ($\exists$) quantifier in F_2.

We can now give Rules 3 and 4 for the definition of a formula we started earlier:

- *Rule 3*: If F is a formula, then so is $(\exists t)(F)$, where t is a tuple variable. The formula $(\exists t)(F)$ is TRUE if the formula F evaluates to TRUE for *some* (at least one) tuple assigned to free occurrences of t in F; otherwise, $(\exists\ t)(F)$ is FALSE.
- *Rule 4*: If F is a formula, then so is $(\forall\ t)(F)$, where t is a tuple variable. The formula $(\forall t)(F)$ is TRUE if the formula F evaluates to TRUE for *every tuple* (in the universe) assigned to free occurrences of t in F; otherwise, $(\forall t)(F)$ is FALSE.

The ($\exists$) quantifier is called an existential quantifier because a formula $(\exists t)(F)$ is TRUE if *there exists* some tuple that makes F TRUE. For the universal quantifier, $(\forall t)(F)$ is TRUE if every possible tuple that can be assigned to free occurrences of t in F is substituted for t, and F is TRUE for *every such substitution*. It is called the universal or *for all* quantifier because every tuple in *the universe of* tuples must make F TRUE to make the quantified formula TRUE.

6.6.4 Example Queries Using the Existential Quantifier

We will use some of the same queries from Section 6.5 to give a flavor of how the same queries are specified in relational algebra and in relational calculus. Notice that some queries are easier to specify in the relational algebra than in the relational calculus, and vice versa.

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

Q1: $\{t.$Fname, $t.$Lname, $t.$Address | EMPLOYEE(t) **AND** $(\exists d)$
(DEPARTMENT(d) **AND** $d.$Dname='Research' **AND** $d.$Dnumber=$t.$Dno)$\}$

The *only free tuple variables* in a relational calculus expression should be those that appear to the left of the bar (|). In Q1, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified in Q1, the attributes Fname, Lname, and ADDRESS are retrieved for each such tuple. The conditions EMPLOYEE(t) and DEPARTMENT(d) specify the range relations for t and d. The condition $d.$Dname = 'Research' is a **selection condition** and corresponds to a SELECT operation in the relational algebra, whereas the condition $d.$Dnumber = $t.$Dno is a **join condition** and serves a similar purpose to the JOIN operation (see Section 6.3).

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

Q2: $\{p.$Pnumber, $p.$Dnum, $m.$Lname, $m.$Bdate, $m.$Address | PROJECT(p)
AND EMPLOYEE(m) **AND** $p.$Plocation='Stafford'
AND $((\exists d)$(DEPARTMENT(d)
AND $p.$Dnum=$d.$Dnumber **AND** $d.$Mgr_ssn=$m.$Ssn))$\}$

In Q2 there are two free tuple variables, p and m. Tuple variable d is bound to the existential quantifier. The query condition is evaluated for every combination of tuples assigned to p and m; and out of all possible combinations of tuples to which p and m are bound, only the combinations that satisfy the condition are selected.

Several tuple variables in a query can range over the same relation. For example, to specify Q8—for each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor—we specify two tuple variables e and s that both range over the EMPLOYEE relation:

Q8: $\{e.$Fname, $e.$Lname, $s.$Fname, $s.$Lname | EMPLOYEE(e) **AND** EMPLOYEE(s)
AND $e.$Super_ssn=$s.$Ssn$\}$

Query 3′. List the name of each employee who works on *some* project controlled by department number 5. This is a variation of Q3 in which *all* is changed to *some*. In this case we need two join conditions and two existential quantifiers.

Q3′: $\{e.$Lname, $e.$Fname | EMPLOYEE(e)
AND $((\exists x)(\exists w)$(PROJECT(x) **AND** WORKS_ON(w) **AND** $x.$Dnum=5
AND $w.$Essn=$e.$Ssn **AND** $x.$Pnumber=$w.$Pno))$\}$

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 manager of the controlling department for the project.

Q4: $\{p.\text{Pnumber} \mid \text{PROJECT}(p)$ **AND** $(\ ((\exists\, e)(\exists\, w)(\text{EMPLOYEE}(e)$
AND $\text{WORKS\_ON}(w)$ **AND** $w.\text{Pno}=p.\text{Pnumber}$
AND $e.\text{Lname}=\text{'Smith'}$ **AND** $e.\text{Ssn}=w.\text{Essn})\ )$
OR
$((\exists\, m)(\exists\, d)(\text{EMPLOYEE}(m)$ **AND** $\text{DEPARTMENT}(d)$
AND $p.\text{Dnum}=d.\text{Dnumber}$ **AND** $d.\text{Mgr\_ssn}=m.\text{Ssn}$
AND $m.\text{Lname}=\text{'Smith'})))\}$

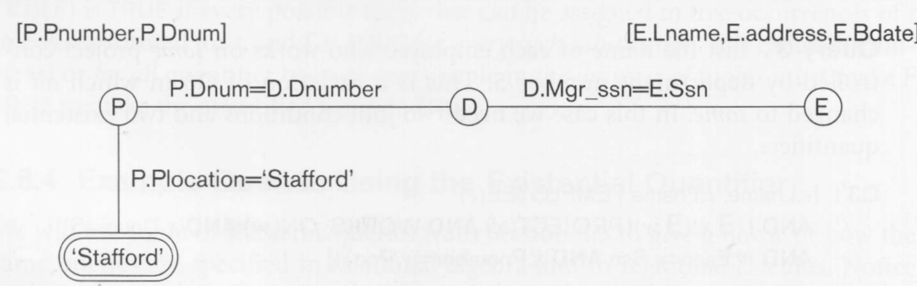
Compare this with the relational algebra version of this query in Section 6.5. The UNION operation in relational algebra can usually be substituted with an OR connective in relational calculus. In the next section we discuss the relationship between the universal and existential quantifiers and show how one can be transformed into the other.

6.6.5 Notation for Query Graphs

In this section we describe a notation that has been proposed to represent relational calculus queries internally. This more neutral representation of a query is called a **query graph**. Figure 6.13 shows the query graph for Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges**, as shown in Figure 6.13. Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

The query graph representation does not include an order on which operations to perform first. There is only a single graph corresponding to each query. Although some optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

Figure 6.13
Query graph for Q2.



6.6.6 Transforming the Universal and Existential Quantifiers

Next, we introduce some well-known transformations from mathematical logic that relate the universal and existential quantifiers. It is possible to transform a universal quantifier into an existential quantifier, and vice versa, to get an equivalent expression. One general transformation can be described informally as follows: Transform one type of quantifier into the other with negation (preceded by **NOT**); **AND** and **OR** replace one another; a negated formula becomes unnegated; and an unnegated formula becomes negated. Some special cases of this transformation can be stated as follows, where the ≡ symbol stands for **equivalent to**:

$(\forall x)\ (P(x)) \equiv \textbf{NOT}\ (\exists x)\ (\textbf{NOT}\ (P(x)))$

$(\exists x)\ (P(x)) \equiv \textbf{NOT}\ (\forall x)\ (\textbf{NOT}\ (P(x)))$

$(\forall x)\ (P(x)\ \textbf{AND}\ Q(x)) \equiv \textbf{NOT}\ (\exists x)\ (\textbf{NOT}\ (P(x))\ \textbf{OR NOT}\ (Q(x)))$

$(\forall x)\ (P(x)\ \textbf{OR}\ Q(x)) \equiv \textbf{NOT}\ (\exists x)\ (\textbf{NOT}\ (P(x))\ \textbf{AND NOT}\ (Q(x)))$

$(\exists x)\ (P(x))\ \textbf{OR}\ Q(x)) \equiv \textbf{NOT}\ (\forall x)\ (\textbf{NOT}\ (P(x))\ \textbf{AND NOT}\ (Q(x)))$

$(\exists x)\ (P(x)\ \textbf{AND}\ Q(x)) \equiv \textbf{NOT}\ (\forall x)\ (\textbf{NOT}\ (P(x))\ \textbf{OR NOT}\ (Q(x)))$

Notice also that the following is TRUE, where the ⇒ symbol stands for **implies:**

$(\forall x)\ (P(x)) \Rightarrow (\exists x)\ (P(x))$

$\textbf{NOT}\ (\exists x)\ (P(x)) \Rightarrow \textbf{NOT}\ (\forall x)\ (P(x))$

6.6.7 Using the Universal Quantifier

Whenever we use a universal quantifier, it is quite judicious to follow a few rules to ensure that our expression makes sense. We discuss these rules with respect to Q3.

Query 3. List the names of employees who work on *all* the projects controlled by department number 5. One way to specify this query is to use the universal quantifier as shown:

Q3: {*e*.Lname, *e*.Fname | EMPLOYEE(*e*) **AND** ((∀ *x*)(**NOT**(PROJECT(*x*))
OR NOT (*x*.Dnum=5) **OR** ((∃ *w*)(WORKS_ON(*w*) **AND** *w*.Essn= *e*.Ssn
AND *x*.Pnumber=*w*.Pno)))))}

We can break up Q3 into its basic components as follows:

Q3: {*e*.Lname, *e*.Fname | EMPLOYEE(*e*) **AND** F'}
F'= ((∀ *x*)(**NOT**(PROJECT(*x*)) **OR** F_1))
F_1 = **NOT**(*x*.Dnum=5) **OR** F_2
F_2 = ((∃ *w*)(WORKS_ON(*w*) **AND** *w*.Essn= *e*.Ssn
AND *x*.Pnumber= *w*.Pno))

We want to make sure that a selected employee *e* works on *all the projects* controlled by department 5, but the definition of universal quantifier says that to make the quantified formula TRUE, the inner formula must be TRUE *for all tuples in the uni-*

verse. The trick is to exclude from the universal quantification all tuples that we are not interested in by making the condition TRUE *for all such tuples*. This is necessary because a universally quantified tuple variable, such as x in Q3, must evaluate to TRUE *for every possible tuple* assigned to it to make the quantified formula TRUE. The first tuples to exclude (by making them evaluate automatically to TRUE) are those that are not in the relation R of interest. In Q3, using the expression **NOT**(PROJECT(x)) inside the universally quantified formula evaluates to TRUE all tuples x that are not in the PROJECT relation. Then we exclude the tuples we are not interested in from R itself. In Q3, using the expression **NOT**(x.Dnum=5) evaluates to TRUE all tuples x that are in the PROJECT relation but are not controlled by department 5. Finally, we specify a condition F_2 that must hold on all the remaining tuples in R. Hence, we can explain Q3 as follows:

1. For the formula $F' = (\forall x)(F)$ to be TRUE, we must have the formula F be TRUE *for all tuples in the universe that can be assigned to x*. However, in Q3 we are only interested in F being TRUE for all tuples of the PROJECT relation that are controlled by department 5. Hence, the formula F is of the form (**NOT**(PROJECT(x)) **OR** F_1). The '**NOT**(PROJECT(x)) **OR** . . .' condition is TRUE for all tuples *not in the PROJECT relation* and has the effect of eliminating these tuples from consideration in the truth value of F_1. For every tuple in the PROJECT relation, F_1 must be TRUE if F' is to be TRUE.
2. Using the same line of reasoning, we do not want to consider tuples in the PROJECT relation that are not controlled by department number 5, since we are only interested in PROJECT tuples whose Dnum=5. Therefore, we can write:

   IF (x.Dnum=5) **THEN** F_2

   which is equivalent to

   (**NOT** (x.Dnum=5) **OR** F_2)
3. Formula F_1, hence, is of the form **NOT**(x.Dnum=5) **OR** F_2. In the context of Q3, this means that, for a tuple x in the PROJECT relation, either its Dnum≠5 or it must satisfy F_2.
4. Finally, F_2 gives the condition that we want to hold for a selected EMPLOYEE tuple: that the employee works on *every* PROJECT *tuple that has not been excluded yet*. Such employee tuples are selected by the query.

In English, Q3 gives the following condition for selecting an EMPLOYEE tuple e: For every tuple x in the PROJECT relation with x.Dnum = 5, there must exist a tuple w in WORKS_ON such that w.Essn = e.Ssn and w.Pno = x.Pnumber. This is equivalent to saying that EMPLOYEE e works on every PROJECT x in DEPARTMENT number 5. (Whew!)

Using the general transformation from universal to existential quantifiers given in Section 6.6.6, we can rephrase the query in Q3 as shown in Q3A:

Q3A: {e.Lname, e.Fname | EMPLOYEE(e) **AND** (**NOT** ($\exists$ x) (PROJECT(x)
AND (x.Dnum=5) **AND**(**NOT** ($\exists$ w)(WORKS_ON(w)
AND w.Essn= e.Ssn **AND** x.Pnumber=w.Pno))))}

We now give some additional examples of queries that use quantifiers.

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

Q6: {*e*.Fname, *e*.Lname | EMPLOYEE(*e*) **AND** (**NOT** (∃ *d*)(DEPENDENT(*d*) **AND** *e*.Ssn=*d*.Essn))}

Using the general transformation rule, we can rephrase Q6 as follows:

Q6A: {*e*.Fname, *e*.Lname | EMPLOYEE(*e*) **AND** ((∀ *d*)(**NOT**(DEPENDENT(*d*)) **OR NOT**(*e*.Ssn=*d*.Essn)))}

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

Q7: {*e*.Fname, *e*.Lname | EMPLOYEE(*e*) **AND** ((∃ *d*)(∃ ρ)(DEPARTMENT(*d*) **AND** DEPENDENT(ρ) **AND** *e*.Ssn=*d*.Mgr_ssn **AND** ρ.Essn=*e*.Ssn))}

This query is handled by interpreting *managers who have at least one dependent* as *managers for whom there exists some dependent*.

6.6.8 Safe Expressions

Whenever we use universal quantifiers, existential quantifiers, or negation of predicates in a calculus expression, we must make sure that the resulting expression makes sense. A **safe expression** in relational calculus is one that is guaranteed to yield a *finite number of tuples* as its result; otherwise, the expression is called **unsafe**. For example, the expression

{*t* | **NOT** (EMPLOYEE(*t*))}

is *unsafe* because it yields all tuples in the universe that are *not* EMPLOYEE tuples, which are infinitely numerous. If we follow the rules for Q3 discussed earlier, we will get a safe expression when using universal quantifiers. We can define safe expressions more precisely by introducing the concept of the *domain of a tuple relational calculus expression:* This is the set of all values that either appear as constant values in the expression or exist in any tuple in the relations referenced in the expression. The domain of {*t* | **NOT**(EMPLOYEE(*t*))} is the set of all attribute values appearing in some tuple of the EMPLOYEE relation (for any attribute). The domain of the expression Q3A would include all values appearing in EMPLOYEE, PROJECT, and WORKS_ON (unioned with the value 5 appearing in the query itself).

An expression is said to be **safe** if all values in its result are from the domain of the expression. Notice that the result of {*t* | **NOT**(EMPLOYEE(*t*))} is unsafe, since it will, in general, include tuples (and hence values) from outside the EMPLOYEE relation; such values are not in the domain of the expression. All of our other examples are safe expressions.

6.7 The Domain Relational Calculus

There is another type of relational calculus called the domain relational calculus, or simply, **domain calculus**. While SQL (see Chapter 8), a language based on tuple relational calculus, was being developed by IBM Research at San Jose, California, another language called QBE (Query-By-Example), which is related to domain calculus, was being developed almost concurrently at IBM T.J. Watson Research Center at Yorktown Heights, New York. The formal specification of the domain calculus was proposed after the development of the QBE system.

Domain calculus differs from tuple calculus in the *type of variables* used in formulas: Rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree n for a query result, we must have n of these **domain variables**—one for each attribute. An expression of the domain calculus is of the form

$$\{x_1, x_2, \ldots, x_n \mid \text{COND}(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m})\}$$

where $x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ are domain variables that range over domains (of attributes), and COND is a **condition** or **formula** of the domain relational calculus.

A formula is made up of **atoms**. The atoms of a formula are slightly different from those for the tuple calculus and can be one of the following:

1. An atom of the form $R(x_1, x_2, \ldots, x_j)$, where R is the name of a relation of degree j and each x_i, $1 \le i \le j$, is a domain variable. This atom states that a list of values of $<x_1, x_2, \ldots, x_j>$ must be a tuple in the relation whose name is R, where x_i is the value of the ith attribute value of the tuple. To make a domain calculus expression more concise, we can *drop the commas* in a list of variables; thus, we can write

   $\{x_1, x_2, \ldots, x_n \mid R(x_1\ x_2\ x_3) \text{ AND} \ldots\}$

   instead of

   $\{x_1, x_2, \ldots, x_n \mid R(x_1, x_2, x_3) \text{ AND} \ldots\}$

2. An atom of the form x_i **op** x_j, where **op** is one of the comparison operators in the set $\{=, <, \le, >, \ge, \ne\}$, and x_i and x_j are domain variables.

3. An atom of the form x_i **op** c or c **op** x_j, where **op** is one of the comparison operators in the set $\{=, <, \le, >, \ge, \ne\}$, x_i and x_j are domain variables, and c is a constant value.

As in tuple calculus, atoms evaluate to either TRUE or FALSE for a specific set of values, called the **truth values** of the atoms. In case 1, if the domain variables are assigned values corresponding to a tuple of the specified relation R, then the atom is TRUE. In cases 2 and 3, if the domain variables are assigned values that satisfy the condition, then the atom is TRUE.

In a similar way to the tuple relational calculus, formulas are made up of atoms, variables, and quantifiers, so we will not repeat the specifications for formulas here.

Some examples of queries specified in the domain calculus follow. We will use lowercase letters $l, m, n, \ldots, x, y, z$ for domain variables.

Query 0. List the birth date and address of the employee whose name is 'John B. Smith'.

Q0: $\{uv \mid (\exists q)\ (\exists r)\ (\exists s)\ (\exists t)\ (\exists w)\ (\exists x)\ (\exists y)\ (\exists z)$
(EMPLOYEE($qrstuvwxyz$) **AND** q='John' **AND** r='B' **AND** s='Smith')$\}$

We need ten variables for the EMPLOYEE relation, one to range over the domain of each attribute in order. Of the ten variables $q, r, s, \ldots, z$, only u and v are free. We first specify the *requested attributes*, Bdate and Address, by the free domain variables u for BDATE and v for ADDRESS. Then we specify the condition for selecting a tuple following the bar (|)—namely, that the sequence of values assigned to the variables $qrstuvwxyz$ be a tuple of the EMPLOYEE relation and that the values for q (Fname), r (Minit), and s (Lname) be 'John', 'B', and 'Smith', respectively. For convenience, we will quantify only those variables *actually appearing in a condition* (these would be q, r, and s in Q0) in the rest of our examples.[13]

An alternative shorthand notation, used in QBE, for writing this query is to assign the constants 'John', 'B', and 'Smith' directly as shown in Q0A. Here, all variables not appearing to the left of the bar are implicitly existentially quantified:[14]

Q0A: $\{uv \mid$ EMPLOYEE('John','B','Smith',t,u,v,w,x,y,z) $\}$

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

Q1: $\{qsv \mid (\exists z)\ (\exists l)\ (\exists m)$ (EMPLOYEE($qrstuvwxyz$) **AND**
DEPARTMENT($lmno$) **AND** l='Research' **AND** $m=z$)$\}$

A condition relating two domain variables that range over attributes from two relations, such as $m = z$ in Q1, is a **join condition**; whereas a condition that relates a domain variable to a constant, such as l = 'Research', is a **selection condition.**

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

Q2: $\{iksuv \mid (\exists j)(\exists m)(\exists n)(\exists t)$(PROJECT($hijk$)
AND EMPLOYEE($qrstuvwxyz$) **AND** DEPARTMENT($lmno$)
AND $k=m$ **AND** $n=t$ **AND** j='Stafford')$\}$

13. Note that the notation of quantifying only the domain variables actually used in conditions and of showing a predicate such as EMPLOYEE(*qrstuvwxyz*) without separating domain variables with commas is an abbreviated notation used for convenience; it is not the correct formal notation.

14. Again, this is not formally accurate notation.

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

Q6: $\{qs \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz)$
AND $(\textbf{NOT}(\exists l)(\text{DEPENDENT}(lmnop)$ **AND** $t{=}l)))\}$

Q6 can be restated using universal quantifiers instead of the existential quantifiers, as shown in Q6A:

Q6A: $\{qs \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz)$
AND $((\forall l)(\textbf{NOT}(\text{DEPENDENT}(lmnop))$ **OR NOT**$(t{=}l))))\}$

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

Q7: $\{sq \mid (\exists t)(\exists j)(\exists l)(\text{EMPLOYEE}(qrstuvwxyz)$ **AND** $\text{DEPARTMENT}(hijk)$
AND $\text{DEPENDENT}(lmnop)$ **AND** $t{=}j$ **AND** $l{=}t)\}$

As we mentioned earlier, it can be shown that any query that can be expressed in the relational algebra can also be expressed in the domain or tuple relational calculus. Also, any *safe expression* in the domain or tuple relational calculus can be expressed in the relational algebra.

The QBE language was based on the domain relational calculus, although this was realized later, after the domain calculus was formalized. QBE was one of the first graphical query languages with minimum syntax developed for database systems. It was developed at IBM Research and is available as an IBM commercial product as part of the Query Management Facility (QMF) interface option to DB2. It has been mimicked by several other commercial products. Because of its important place in the field of relational languages, we have included an overview of QBE in Appendix D.

6.8 Summary

In this chapter we presented two formal languages for the relational model of data. They are used to manipulate relations and produce new relations as answers to queries. We discussed the relational algebra and its operations, which are used to specify a sequence of operations to specify a query. Then we introduced two types of relational calculi called tuple calculus and domain calculus; they are declarative in that they specify the result of a query without specifying how to produce the query result.

In Sections 6.1 through 6.3, we introduced the basic relational algebra operations and illustrated the types of queries for which each is used. First, we discussed the unary relational operators SELECT and PROJECT, as well as the RENAME operation. Then, we discussed binary set theoretic operations requiring that relations on which they are applied be union compatible; these include UNION, INTERSECTION, and SET DIFFERENCE. The CARTESIAN PRODUCT operation is a set operation that can be used to combine tuples from two relations, producing all possible combinations.

It is rarely used in practice; however, we showed how CARTESIAN PRODUCT followed by SELECT can be used to define matching tuples from two relations and leads to the JOIN operation. Different JOIN operations called THETA JOIN, EQUIJOIN, and NATURAL JOIN were introduced. Query trees were introduced as an internal representation of relational algebra queries.

We discussed some important types of queries that *cannot* be stated with the basic relational algebra operations but are important for practical situations. We introduced GENERALIZED PROJECTION to use functions of attributes in the projection list and the AGGREGATE FUNCTION operation to deal with aggregate types of requests. We discussed recursive queries, for which there is no direct support in the algebra but which can be approached in a step-by-step approach, as we demonstrated. Then we presented the OUTER JOIN and OUTER UNION operations, which extend JOIN and UNION and allow all information in source relations to be preserved in the result.

The last two sections described the basic concepts behind relational calculus, which is based on the branch of mathematical logic called predicate calculus. There are two types of relational calculi: (1) the tuple relational calculus, which uses tuple variables that range over tuples (rows) of relations, and (2) the domain relational calculus, which uses domain variables that range over domains (columns of relations). In relational calculus, a query is specified in a single declarative statement, without specifying any order or method for retrieving the query result. Hence, relational calculus is often considered to be a higher-level language than the relational algebra because a relational calculus expression states *what* we want to retrieve regardless of *how* the query may be executed.

We discussed the syntax of relational calculus queries using both tuple and domain variables. We introduced query graphs as an internal representation for queries in relational calculus. We also discussed the existential quantifier (∃) and the universal quantifier (∀). We saw that relational calculus variables are bound by these quantifiers. We described in detail how queries with universal quantification are written, and we discussed the problem of specifying safe queries whose results are finite. We also discussed rules for transforming universal into existential quantifiers, and vice versa. It is the quantifiers that give expressive power to the relational calculus, making it equivalent to relational algebra. There is no analog to grouping and aggregation functions in basic relational calculus, although some extensions have been suggested.

Review Questions

6.1. List the operations of relational algebra and the purpose of each.

6.2. What is union compatibility? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible?