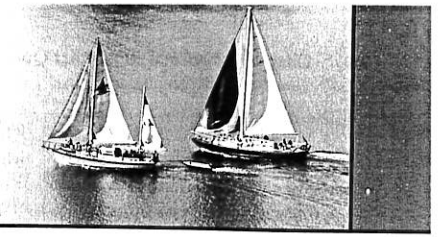
CHAPTER 6



Formal Relational Query Languages

In Chapters 2 through 5 we introduced the relational model and covered SQL in great detail. In this chapter we present the formal model upon which SQL as well as other relational query languages are based.

We cover three formal languages. We start by presenting the relational algebra, which forms the basis of the widely used SQL query language. We then cover the tuple relational calculus and the domain relational calculus, which are declarative query languages based on mathematical logic.

6.1 The Relational Algebra

The relational algebra is a *procedural* query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are *select*, *project*, *union*, *set difference*, *Cartesian product*, and *rename*. In addition to the fundamental operations, there are several other operations—namely, *set intersection*, *natural join*, and *assignment*. We shall define these operations in terms of the fundamental operations.

6.1.1 Fundamental Operations

The select, project, and rename operations are called *unary* operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called *binary* operations.

6.1.1.1 The Select Operation

The **select** operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. Thus, to select

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 6.1 The *instructor* relation.

those tuples of the *instructor* relation where the instructor is in the "Physics" department, we write:

$$\sigma_{dept\_name = \text{"Physics"}}(instructor)$$

If the *instructor* relation is as shown in Figure 6.1, then the relation that results from the preceding query is as shown in Figure 6.2.

We can find all instructors with salary greater than \$90,000 by writing:

$$\sigma_{salary > 90000}(instructor)$$

In general, we allow comparisons using $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). Thus, to find the instructors in Physics with a salary greater than \$90,000, we write:

$$\sigma_{dept\_name = \text{"Physics"} \wedge salary > 90000}(instructor)$$

The selection predicate may include comparisons between two attributes. To illustrate, consider the relation *department*. To find all departments whose name is the same as their building name, we can write:

$$\sigma_{dept\_name = building}(department)$$

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Figure 6.2 Result of $\sigma_{dept\_name = \text{"Physics"}}(instructor)$.

SQL VERSUS RELATIONAL ALGEBRA

The term *select* in relational algebra has a different meaning than the one used in SQL, which is an unfortunate historical fact. In relational algebra, the term *select* corresponds to what we refer to in SQL as *where*. We emphasize the different interpretations here to minimize potential confusion.

6.1.1.2 The Project Operation

Suppose we want to list all instructors' *ID*, *name*, and *salary*, but do not care about the *dept_name*. The **project** operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi (Π). We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses. We write the query to produce such a list as:

$$\Pi_{ID,\ name,\ salary}(instructor)$$

Figure 6.3 shows the relation that results from this query.

6.1.1.3 Composition of Relational Operations

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find the name of all instructors in the Physics department." We write:

ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Figure 6.3 Result of $\Pi_{ID,\ name,\ salary}(instructor)$.

$$\Pi_{name}\ (\sigma_{dept\_name = \text{"Physics"}}\ (instructor))$$

Notice that, instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

In general, since the result of a relational-algebra operation is of the same type (relation) as its inputs, relational-algebra operations can be composed together into a **relational-algebra expression**. Composing relational-algebra operations into relational-algebra expressions is just like composing arithmetic operations (such as $+$, $-$, $*$, and $\div$) into arithmetic expressions. We study the formal definition of relational-algebra expressions in Section 6.1.2.

6.1.1.4 The Union Operation

Consider a query to find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both. The information is contained in the *section* relation (Figure 6.4). To find the set of all courses taught in the Fall 2009 semester, we write:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Fall"} \wedge year = 2009}\ (section))$$

To find the set of all courses taught in the Spring 2010 semester, we write:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Spring"} \wedge year = 2010}\ (section))$$

To answer the query, we need the **union** of these two sets; that is, we need all section IDs that appear in either or both of the two relations. We find these data

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 6.4 The *section* relation.

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 6.5 Courses offered in either Fall 2009, Spring 2010 or both semesters.

by the binary operation union, denoted, as in set theory, by $\cup$. So the expression needed is:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Fall"} \wedge year = 2009}\ (section)) \cup \Pi_{course\_id}\ (\sigma_{semester = \text{"Spring"} \wedge year = 2010}\ (section))$$

The result relation for this query appears in Figure 6.5. Notice that there are 8 tuples in the result, even though there are 3 distinct courses offered in the Fall 2009 semester and 6 distinct courses offered in the Spring 2010 semester. Since relations are sets, duplicate values such as CS-101, which is offered in both semesters, are replaced by a single occurrence.

Observe that, in our example, we took the union of two sets, both of which consisted of *course_id* values. In general, we must ensure that unions are taken between *compatible* relations. For example, it would not make sense to take the union of the *instructor* relation and the *student* relation. Although both relations have four attributes, they differ on the *salary* and *tot_cred* domains. The union of these two attributes would not make sense in most situations. Therefore, for a union operation $r \cup s$ to be valid, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.
2. The domains of the ith attribute of r and the ith attribute of s must be the same, for all i.

Note that r and s can be either database relations or temporary relations that are the result of relational-algebra expressions.

6.1.1.5 The Set-Difference Operation

The **set-difference** operation, denoted by $-$, allows us to find tuples that are in one relation but are not in another. The expression $r - s$ produces a relation containing those tuples in r but not in s.

course_id
CS-347
PHY-101

Figure 6.6 Courses offered in the Fall 2009 semester but not in Spring 2010 semester.

We can find all the courses taught in the Fall 2009 semester but not in Spring 2010 semester by writing:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Fall"} \wedge year=2009}\ (section))\ -$$
$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Spring"} \wedge year=2010}\ (section))$$

The result relation for this query appears in Figure 6.6.

As with the union operation, we must ensure that set differences are taken between *compatible* relations. Therefore, for a set-difference operation $r - s$ to be valid, we require that the relations r and s be of the same arity, and that the domains of the ith attribute of r and the ith attribute of s be the same, for all i.

6.1.1.6 The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross ($\times$), allows us to combine information from any two relations. We write the Cartesian product of relations r_1 and r_2 as $r_1 \times r_2$.

Recall that a relation is by definition a subset of a Cartesian product of a set of domains. From that definition, we should already have an intuition about the definition of the Cartesian-product operation. However, since the same attribute name may appear in both r_1 and r_2, we need to devise a naming schema to distinguish between these attributes. We do so here by attaching to an attribute the name of the relation from which the attribute originally came. For example, the relation schema for $r = instructor \times teaches$ is:

(*instructor.ID, instructor.name, instructor.dept_name, instructor.salary*
teaches.ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

With this schema, we can distinguish *instructor.ID* from *teaches.ID*. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for r as:

(*instructor.ID, name, dept_name, salary*
teaches.ID, course_id, sec_id, semester, year)

This naming convention *requires* that the relations that are the arguments of the Cartesian-product operation have distinct names. This requirement causes problems in some cases, such as when the Cartesian product of a relation with itself is desired. A similar problem arises if we use the result of a relational-algebra expression in a Cartesian product, since we shall need a name for the relation so

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure 6.7 The *teaches* relation.

that we can refer to the relation's attributes. In Section 6.1.1.7, we see how to avoid these problems by using the rename operation.

Now that we know the relation schema for $r = instructor \times teaches$, what tuples appear in r? As you may suspect, we construct a tuple of r out of each possible pair of tuples: one from the *instructor* relation (Figure 6.1) and one from the *teaches* relation (Figure 6.7). Thus, r is a large relation, as you can see from Figure 6.8, which includes only a portion of the tuples that make up r.[1]

Assume that we have n_1 tuples in *instructor* and n_2 tuples in *teaches*. Then, there are $n_1 * n_2$ ways of choosing a pair of tuples—one tuple from each relation; so there are $n_1 * n_2$ tuples in r. In particular, note that for some tuples t in r, it may be that $t[instructor.ID] \neq t[teaches.ID]$.

In general, if we have relations $r_1(R_1)$ and $r_2(R_2)$, then $r_1 \times r_2$ is a relation whose schema is the concatenation of R_1 and R_2. Relation R contains all tuples t for which there is a tuple t_1 in r_1 and a tuple t_2 in r_2 for which $t[R_1] = t_1[R_1]$ and $t[R_2] = t_2[R_2]$.

Suppose that we want to find the names of all instructors in the Physics department together with the *course_id* of all courses they taught. We need the information in both the *instructor* relation and the *teaches* relation to do so. If we write:

$$\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)$$

then the result is the relation in Figure 6.9.

[1] Note that we renamed *instructor.ID* as *inst.ID* to reduce the width of the tables in Figures 6.8 and 6.9.

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Physics	95000	10101	CS-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	CS-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 6.8 Result of *instructor* × *teaches*.

We have a relation that pertains only to instructors in the Physics department. However, the *course_id* column may contain information about courses that were not taught by the corresponding instructor. (If you do not see why that is true, recall that the Cartesian product takes all possible pairings of one tuple from *instructor* with one tuple of *teaches*.)

Since the Cartesian-product operation associates *every* tuple of *instructor* with every tuple of *teaches*, we know that if a an instructor is in the Physics department, and has taught a course (as recorded in the *teaches* relation), then there is some

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
22222	Einstein	Physics	95000	10101	CS-437	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
22222	Einstein	Physics	95000	32343	HIS-351	1	Spring	2010
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
33456	Gold	Physics	87000	10101	CS-437	1	Fall	2009
33456	Gold	Physics	87000	10101	CS-315	1	Spring	2010
33456	Gold	Physics	87000	12121	FIN-201	1	Spring	2010
33456	Gold	Physics	87000	15151	MU-199	1	Spring	2010
33456	Gold	Physics	87000	22222	PHY-101	1	Fall	2009
33456	Gold	Physics	87000	32343	HIS-351	1	Spring	2010
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 6.9 Result of $\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)$.

tuple in $\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)$ that contains his name, and which satisfies $instructor.ID = teaches.ID$. So, if we write:

$$\sigma_{instructor.ID = teaches.ID}(\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches))$$

we get only those tuples of $instructor \times teaches$ that pertain to instructors in Physics and the courses that they taught.

Finally, since we only want the names of all instructors in the Physics department together with the *course_id* of all courses they taught, we do a projection:

$$\Pi_{name,\ course\_id}\ (\sigma_{instructor.ID = teaches.ID}(\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)))$$

The result of this expression, shown in Figure 6.10, is the correct answer to our query. Observe that although instructor Gold is in the Physics department, he does not teach any course (as recorded in the *teaches* relation), and therefore does not appear in the result.

name	*course_id*
Einstein	PHY-101

Figure 6.10 Result of $\Pi_{name,\ course\_id}\ (\sigma_{instructor.ID = teaches.ID}(\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)))$.

Note that there is often more than one way to write a query in relational algebra. Consider the following query:

$$\Pi_{name,\ course\_id}\ (\sigma_{instructor.ID = teaches.ID}((\sigma_{dept\_name = \text{"Physics"}}(instructor)) \times teaches))$$

Note the subtle difference between the two queries: in the query above, the selection that restricts *dept_name* to Physics is applied to *instructor*, and the Cartesian product is applied subsequently; in contrast, the Cartesian product was applied before the selection in the earlier query. However, the two queries are **equivalent**; that is, they give the same result on any database.

6.1.1.7 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter rho (ρ), lets us do this. Given a relational-algebra expression E, the expression

$$\rho_x\ (E)$$

returns the result of expression E under the name x.

A relation r by itself is considered a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation r to get the same relation under a new name.

A second form of the rename operation is as follows: Assume that a relational-algebra expression E has arity n. Then, the expression

$$\rho_{x(A_1, A_2, \ldots, A_n)}\ (E)$$

returns the result of expression E under the name x, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.

To illustrate renaming a relation, we consider the query "Find the highest salary in the university." Our strategy is to (1) compute first a temporary relation consisting of those salaries that are *not* the largest and (2) take the set difference between the relation $\Pi_{salary}\ (instructor)$ and the temporary relation just computed, to obtain the result.

1. Step 1: To compute the temporary relation, we need to compare the values of all salaries. We do this comparison by computing the Cartesian product *instructor* $\times$ *instructor* and forming a selection to compare the value of any two salaries appearing in one tuple. First, we need to devise a mechanism to distinguish between the two *salary* attributes. We shall use the rename operation to rename one reference to the instructor relation; thus we can reference the relation twice without ambiguity.

salary
65000
90000
40000
60000
87000
75000
62000
72000
80000
92000

Figure 6.11 Result of the subexpression $\Pi_{instructor.salary}\ (\sigma_{instructor.salary\ <\ d.salary}\ (instructor \times \rho_d\ (instructor)))$.

We can now write the temporary relation that consists of the salaries that are not the largest:

$$\Pi_{instructor.salary}\ (\sigma_{instructor.salary\ <\ d.salary}\ (instructor \times \rho_d\ (instructor)))$$

This expression gives those salaries in the *instructor* relation for which a larger salary appears somewhere in the *instructor* relation (renamed as d). The result contains all salaries *except* the largest one. Figure 6.11 shows this relation.

2. Step 2: The query to find the largest salary in the university can be written as:

$$\Pi_{salary}\ (instructor) - \Pi_{instructor.salary}\ (\sigma_{instructor.salary\ <\ d.salary}\ (instructor \times \rho_d\ (instructor)))$$

Figure 6.12 shows the result of this query.

The rename operation is not strictly required, since it is possible to use a positional notation for attributes. We can name attributes of a relation implicitly by using a positional notation, where \$1, \$2, ... refer to the first attribute, the second attribute, and so on. The positional notation also applies to results of relational-algebra operations. The following relational-algebra expression illustrates the

salary
95000

Figure 6.12 Highest salary in the university.

use of positional notation to write the expression we saw earlier, which computes salaries that are not the largest:

$$\Pi_{\$4}\ (\sigma_{\$4\ <\ \$8}\ (instructor\ \times\ instructor))$$

Note that the Cartesian product concatenates the attributes of the two relations. Thus, for the result of the Cartesian product (*instructor* × *instructor*), \$4 refers to the *salary* attribute from the first occurrence of *instructor*, while \$8 refers to the *salary* attribute from the second occurrence of *instructor*. A positional notation can also be used to refer to relation names, if a binary operation needs to distinguish between its two operand relations. For example, $\$R1$ could refer to the first operand relation, and $\$R2$ could refer to the second operand relation of a Cartesian product. However, the positional notation is inconvenient for humans, since the position of the attribute is a number, rather than an easy-to-remember attribute name. Hence, we do not use the positional notation in this textbook.

6.1.2 Formal Definition of the Relational Algebra

The operations in Section 6.1.1 allow us to give a complete definition of an expression in the relational algebra. A basic expression in the relational algebra consists of either one of the following:

- A relation in the database
- A constant relation

A constant relation is written by listing its tuples within { }, for example { (22222, Einstein, Physics, 95000), (76543, Singh, Finance, 80000) }.

A general expression in the relational algebra is constructed out of smaller subexpressions. Let E_1 and E_2 be relational-algebra expressions. Then, the following are all relational-algebra expressions:

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, where P is a predicate on attributes in E_1
- $\Pi_S(E_1)$, where S is a list consisting of some of the attributes in E_1
- $\rho_x\ (E_1)$, where x is the new name for the result of E_1

6.1.3 Additional Relational-Algebra Operations

The fundamental operations of the relational algebra are sufficient to express any relational-algebra query. However, if we restrict ourselves to just the fundamental operations, certain common queries are lengthy to express. Therefore, we define additional operations that do not add any power to the algebra, but simplify

course_id
CS-101

Figure 6.13 Courses offered in both the Fall 2009 and Spring 2010 semesters.

common queries. For each new operation, we give an equivalent expression that uses only the fundamental operations.

6.1.3.1 The Set-Intersection Operation

The first additional relational-algebra operation that we shall define is **set intersection** (∩). Suppose that we wish to find the set of all courses taught in both the Fall 2009 and the Spring 2010 semesters. Using set intersection, we can write

$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Fall"} \wedge year=2009}\ (section)) \cap \Pi_{course\_id}\ (\sigma_{semester = \text{"Spring"} \wedge year=2010}\ (section))$$

The result relation for this query appears in Figure 6.13.

Note that we can rewrite any relational-algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:

$$r \cap s = r - (r - s)$$

Thus, set intersection is not a fundamental operation and does not add any power to the relational algebra. It is simply more convenient to write $r \cap s$ than to write $r - (r - s)$.

6.1.3.2 The Natural-Join Operation

It is often desirable to simplify certain queries that require a Cartesian product. Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product. The selection operation most often requires that all attributes that are common to the relations that are involved in the Cartesian product be equated.

In our example query from Section 6.1.1.6 that combined information from the *instructor* and *teaches* tables, the matching condition required *instructor.ID* to be equal to *teaches.ID*. These are the only attributes in the two relations that have the same name.

The *natural join* is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the **join** symbol ⋈. The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes. Returning to the example of the relations *instructor* and *teaches*, computing *instructor* **natural join** *teaches* considers only those pairs of tuples where both the tuple from *instructor* and the

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 6.14 The natural join of the *instructor* relation with the *teaches* relation.

tuple from *teaches* have the same value on the common attribute *ID*. The result relation, shown in Figure 6.14, has only 13 tuples, the ones that give information about an instructor and a course that that instructor actually teaches. Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once. Notice also the order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.

Although the definition of natural join is complicated, the operation is easy to apply. As an illustration, consider again the example "Find the names of all instructors together with the *course_id* of all courses they taught." We express this query by using the natural join as follows:

$$\Pi_{name,\ course\_id}\ (instructor \bowtie teaches)$$

Since the schemas for *instructor* and *teaches* have the attribute *ID* in common, the natural-join operation considers only pairs of tuples that have the same value on *ID*. It combines each such pair of tuples into a single tuple on the union of the two schemas; that is, (*ID*, *name*, *dept_name*, *salary*, *course_id*). After performing the projection, we obtain the relation in Figure 6.15.

Consider two relation schemas R and S—which are, of course, lists of attribute names. If we consider the schemas to be *sets*, rather than lists, we can denote those attribute names that appear in both R and S by $R \cap S$, and denote those attribute names that appear in R, in S, or in both by $R \cup S$. Similarly, those attribute names that appear in R but not S are denoted by $R - S$, whereas $S - R$ denotes those

name	*course_id*
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 6.15 Result of $\Pi_{name,\ course\_id}\ (instructor \bowtie teaches)$.

attribute names that appear in S but not in R. Note that the union, intersection, and difference operations here are on sets of attributes, rather than on relations.

We are now ready for a formal definition of the natural join. Consider two relations $r(R)$ and $s(S)$. The **natural join** of r and s, denoted by $r \bowtie s$, is a relation on schema $R \cup S$ formally defined as follows:

$$r \bowtie s = \Pi_{R \cup S}\ (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}\ (r \times s))$$

where $R \cap S = \{A_1, A_2, \ldots, A_n\}$.

Please note that if $r(R)$ and $s(S)$ are relations without any attributes in common, that is, $R \cap S = \emptyset$, then $r \bowtie s = r \times s$.

Let us consider one more example of the use of natural join, to write the query "Find the names of all instructors in the Comp. Sci. department together with the course titles of all the courses that the instructors teach."

$$\Pi_{name,title}\ (\sigma_{dept\_name = \text{"Comp. Sci."}}\ (instructor \bowtie teaches \bowtie course))$$

The result relation for this query appears in Figure 6.16.

Notice that we wrote *instructor* ⋈ *teaches* ⋈ *course* without inserting parentheses to specify the order in which the natural-join operations on the three relations should be executed. In the preceding case, there are two possibilities:

$$(instructor \bowtie teaches) \bowtie course$$
$$instructor \bowtie (teaches \bowtie course)$$

We did not specify which expression we intended, because the two are equivalent. That is, the natural join is associative.

name	*title*
Brandt	Game Design
Brandt	Image Processing
Katz	Image Processing
Katz	Intro. to Computer Science
Srinivasan	Intro. to Computer Science
Srinivasan	Robotics
Srinivasan	Database System Concepts

Figure 6.16 Result of $\Pi_{name,title}(\sigma_{dept\_name = \text{"Comp. Sci."}}(instructor \bowtie teaches \bowtie course))$.

The *theta join* operation is a variant of the natural-join operation that allows us to combine a selection and a Cartesian product into a single operation. Consider relations $r(R)$ and $s(S)$, and let θ be a predicate on attributes in the schema $R \cup S$. The **theta join** operation $r \bowtie_\theta s$ is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

6.1.3.3 The Assignment Operation

It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables. The **assignment** operation, denoted by $\leftarrow$, works like assignment in a programming language. To illustrate this operation, consider the definition of the natural-join operation. We could write $r \bowtie s$ as:

$$\begin{aligned} temp1 &\leftarrow R \times S \\ temp2 &\leftarrow \sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}(temp1) \\ result &= \Pi_{R \cup S}(temp2) \end{aligned}$$

The evaluation of an assignment does not result in any relation being displayed to the user. Rather, the result of the expression to the right of the $\leftarrow$ is assigned to the relation variable on the left of the $\leftarrow$. This relation variable may be used in subsequent expressions.

With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query. For relational-algebra queries, assignment must always be made to a temporary relation variable. Assignments to permanent relations constitute a database modification. Note that the assignment operation does not provide any additional power to the algebra. It is, however, a convenient way to express complex queries.

6.1.3.4 Outer join Operations

The **outer-join** operation is an extension of the join operation to deal with missing information. Suppose that there is some instructor who teaches no courses. Then

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
33456	Gold	Physics	87000	*null*	*null*	*null*	*null*
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
58583	Califieri	History	62000	*null*	*null*	*null*	*null*
76543	Singh	Finance	80000	*null*	*null*	*null*	*null*
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 6.17 Result of $instructor \bowtie teaches$.

the tuple in the *instructor* relation (Figure 6.1) for that particular instructor would not satisfy the condition of a natural join with the *teaches* relation (Figure 6.7) and that instructor's data would not appear in the result of the natural join, shown in Figure 6.14. For example, instructors Califieri, Gold, and Singh do not appear in the result of the natural join, since they do not teach any course.

More generally, some tuples in either or both of the relations being joined may be "lost" in this way. The **outer join** operation works in a manner similar to the natural join operation we have already studied, but preserves those tuples that would be lost in an join by creating tuples in the result containing null values.

We can use the *outer-join* operation to avoid this loss of information. There are actually three forms of the operation: *left outer join*, denoted $\bowtie$; *right outer join*, denoted $\bowtie$; and *full outer join*, denoted $\bowtie$. All three forms of outer join compute the join, and add extra tuples to the result of the join. For example, the results of the expression $instructor \bowtie teaches$ and $teaches \bowtie instructor$ appear in Figures 6.17 and 6.18, respectively.

The **left outer join** ($\bowtie$) takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join. In Figure 6.17, tuple (58583, Califieri, History, 62000, *null*, *null*, *null*, *null*), is such a tuple. All information from the left relation is present in the result of the left outer join.

ID	*course_id*	*sec_id*	*semester*	*year*	*name*	*dept_name*	*salary*
10101	CS-101	1	Fall	2009	Srinivasan	Comp. Sci.	65000
10101	CS-315	1	Spring	2010	Srinivasan	Comp. Sci.	65000
10101	CS-347	1	Fall	2009	Srinivasan	Comp. Sci.	65000
12121	FIN-201	1	Spring	2010	Wu	Finance	90000
15151	MU-199	1	Spring	2010	Mozart	Music	40000
22222	PHY-101	1	Fall	2009	Einstein	Physics	95000
32343	HIS-351	1	Spring	2010	El Said	History	60000
33456	*null*	*null*	*null*	*null*	Gold	Physics	87000
45565	CS-101	1	Spring	2010	Katz	Comp. Sci.	75000
45565	CS-319	1	Spring	2010	Katz	Comp. Sci.	75000
58583	*null*	*null*	*null*	*null*	Califieri	History	62000
76543	*null*	*null*	*null*	*null*	Singh	Finance	80000
76766	BIO-101	1	Summer	2009	Crick	Biology	72000
76766	BIO-301	1	Summer	2010	Crick	Biology	72000
83821	CS-190	1	Spring	2009	Brandt	Comp. Sci.	92000
83821	CS-190	2	Spring	2009	Brandt	Comp. Sci.	92000
83821	CS-319	2	Spring	2010	Brandt	Comp. Sci.	92000
98345	EE-181	1	Spring	2009	Kim	Elec. Eng.	80000

Figure 6.18 Result of *teaches* ⟕ *instructor*.

The **right outer join** (⟖) is symmetric with the left outer join: It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join. In Figure 6.18, tuple (58583, *null*, *null*, *null*, *null*, Califieri, History, 62000), is such a tuple. Thus, all information from the right relation is present in the result of the right outer join.

The **full outer join**(⟗) does both the left and right outer join operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join.

Note that in going from our left-outer-join example to our right-outer-join example, we chose to swap the order of the operands. Thus both examples preserve tuples from the *instructor* relation, and thus contain the same information. In our example relations, *teaches* tuples always have matching *instructor* tuples, and thus *teaches* ⟗ *instructor* would give the same result as *teaches* ⋈ *instructor*. If there were tuples in *teaches* without matching tuples in *instructor*, such tuples would appear padded with nulls in *teaches* ⟗ *instructor* as well as in *teaches* ⟕ *instructor*. Further examples of outer joins (expressed in SQL syntax) may be found in Section 4.1.2.

Since outer-join operations may generate results containing null values, we need to specify how the different relational-algebra operations deal with null values. Section 3.6 dealt with this issue in the context of SQL. The same concepts apply for the case of relational algebra, and we omit details.

It is interesting to note that the outer-join operations can be expressed by the basic relational-algebra operations. For instance, the left outer join operation, $r \text{ ⟕ } s$, can be written as:

$$(r \bowtie s) \cup (r - \Pi_R(r \bowtie s)) \times \{(null, \ldots, null)\}$$

where the constant relation $\{(null, \ldots, null)\}$ is on the schema $S - R$.

6.1.4 Extended Relational-Algebra Operations

We now describe relational-algebra operations that provide the ability to write queries that cannot be expressed using the basic relational-algebra operations. These operations are called **extended relational-algebra** operations.

6.1.4.1 Generalized Projection

The first operation is the **generalized-projection** operation, which extends the projection operation by allowing operations such as arithmetic and string functions to be used in the projection list. The generalized-projection operation has the form:

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

where E is any relational-algebra expression, and each of $F_1, F_2, \ldots, F_n$ is an arithmetic expression involving constants and attributes in the schema of E. As a base case, the expression may be simply an attribute or a constant. In general, an expression can use arithmetic operations such as $+$, $-$, $*$, and $\div$ on numeric valued attributes, numeric constants, and on expressions that generate a numeric result. Generalized projection also permits operations on other data types, such as concatenation of strings.

For example, the expression:

$$\Pi_{ID, name, dept\_name, salary \div 12}(instructor)$$

gives the *ID*, *name*, *dept_name*, and the monthly salary of each instructor.

6.1.4.2 Aggregation

The second extended relational-algebra operation is the aggregate operation $\mathcal{G}$, which permits the use of aggregate functions such as min or average, on sets of values.

Aggregate functions take a collection of values and return a single value as a result. For example, the aggregate function **sum** takes a collection of values and returns the sum of the values. Thus, the function **sum** applied on the collection:

$$\{1, 1, 3, 4, 4, 11\}$$

returns the value 24. The aggregate function **avg** returns the average of the values. When applied to the preceding collection, it returns the value 4. The aggregate function **count** returns the number of the elements in the collection, and returns 6 on the preceding collection. Other common aggregate functions include **min** and **max**, which return the minimum and maximum values in a collection; they return 1 and 11, respectively, on the preceding collection.

The collections on which aggregate functions operate can have multiple occurrences of a value; the order in which the values appear is not relevant. Such collections are called **multisets**. Sets are a special case of multisets where there is only one copy of each element.

To illustrate the concept of aggregation, we shall use the *instructor* relation. Suppose that we want to find out the sum of salaries of all instructors; the relational-algebra expression for this query is:

$$\mathcal{G}_{\mathbf{sum}(salary)}(instructor)$$

The symbol $\mathcal{G}$ is the letter G in calligraphic font; read it as "calligraphic G." The relational-algebra operation $\mathcal{G}$ signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied. The result of the expression above is a relation with a single attribute, containing a single row with a numerical value corresponding to the sum of the salaries of all instructors.

There are cases where we must eliminate multiple occurrences of a value before computing an aggregate function. If we do want to eliminate duplicates, we use the same function names as before, with the addition of the hyphenated string "**distinct**" appended to the end of the function name (for example, **count-distinct**). An example arises in the query "Find the total number of instructors who teach a course in the Spring 2010 semester." In this case, an instructor counts only once, regardless of the number of course sections that the instructor teaches. The required information is contained in the relation *teaches*, and we write this query as follows:

$$\mathcal{G}_{\mathbf{count-distinct}(ID)}(\sigma_{semester=\text{"Spring"} \wedge year\ =2010}(teaches))$$

The aggregate function **count-distinct** ensures that even if an instructor teaches more than one course, she is counted only once in the result.

There are circumstances where we would like to apply the aggregate function not to a single set of tuples, but instead to a group of sets of tuples. As an illustration, consider the query "Find the average salary in each department." We write this query as follows:

$$_{dept\_name}\mathcal{G}_{\mathbf{average}(salary)}(instructor)$$

Figure 6.19 shows the tuples in the *instructor* relation grouped by the *dept_name* attribute. This is the first step in computing the query result. The specified aggregate is computed for each group, and the result of the query is shown in Figure 6.20.

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

Figure 6.19 Tuples of the *instructor* relation, grouped by the *dept_name* attribute

In contrast, consider the query "Find the average salary of all instructors." We write this query as follows:

$$\mathcal{G}_{\mathbf{average}(salary)}(instructor)$$

In this case the attribute *dept_name* has been omitted from the left side of the $\mathcal{G}$ operator, so the entire relation is treated as a single group.

The general form of the **aggregation operation** $\mathcal{G}$ is as follows:

$$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),\ F_2(A_2),\ldots,\ F_m(A_m)}(E)$$

where E is any relational-algebra expression; $G_1, G_2, \ldots, G_n$ constitute a list of attributes on which to group; each F_i is an aggregate function; and each A_i is an attribute name. The meaning of the operation is as follows: The tuples in the result of expression E are partitioned into groups in such a way that:

dept_name	*salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 6.20 The result relation for the query "Find the average salary in each department".

MULTISET RELATIONAL ALGEBRA

Unlike the relational algebra, SQL allows multiple copies of a tuple in an input relation as well as in a query result. The SQL standard defines how many copies of each tuple are there in the output of a query, which depends in turn on how many copies of tuples are present in the input relations.

To model this behavior of SQL, a version of relational algebra, called the **multiset relational algebra**, is defined to work on multisets, that is, sets that may contain duplicates. The basic operations in the multiset relational algebra are defined as follows:

1. If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
2. For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$, where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
3. If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 * c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$.

For example, suppose that relations r_1 with schema (A, B) and r_2 with schema (C) are the following multisets:

$$r_1 = \{(1, a), (2, a)\} \qquad r_2 = \{(2), (3), (3)\}$$

Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, whereas $\Pi_B(r_1) \times r_2$ would be:

$$\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$$

Multiset union, intersection and set difference can also be defined in a similar way, following the corresponding definitions in SQL, which we saw in Section 3.5. There is no change in the definition of the aggregation operation.

1. All tuples in a group have the same values for $G_1, G_2, \ldots, G_n$.
2. Tuples in different groups have different values for $G_1, G_2, \ldots, G_n$.

Thus, the groups can be identified by the values of attributes $G_1, G_2, \ldots, G_n$. For each group $(g_1, g_2, \ldots, g_n)$, the result has a tuple $(g_1, g_2, \ldots, g_n, a_1, a_2, \ldots, a_m)$ where, for each i, a_i is the result of applying the aggregate function F_i on the multiset of values for attribute A_i in the group.

As a special case of the aggregate operation, the list of attributes $G_1, G_2, \ldots, G_n$ can be empty, in which case there is a single group containing all tuples in the relation. This corresponds to aggregation without grouping.

SQL AND RELATIONAL ALGEBRA

From a comparison of the relational algebra operations and the SQL operations, it should be clear that there is a close connection between the two. A typical SQL query has the form:

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P

Each A_i represents an attribute, and each r_i a relation. P is a predicate. The query is equivalent to the multiset relational-algebra expression:

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m))$$

If the **where** clause is omitted, the predicate P is **true**.

More complex SQL queries can also be rewritten in relational algebra. For example, the query:

select A_1, A_2, **sum**(A_3)
from $r_1, r_2, \ldots, r_m$
where P
group by A_1, A_2

is equivalent to:

$$A_1, A_2\mathcal{G}_{\text{sum}(A_3)}(\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m)))$$

Join expressions in the **from** clause can be written using equivalent join expressions in relational algebra; we leave the details as an exercise for the reader. However, subqueries in the **where** or **select** clause cannot be rewritten into relational algebra in such a straightforward manner, since there is no relational-algebra operation equivalent to the subquery construct. Extensions of relational algebra have been proposed for this task, but are beyond the scope of this book.

6.2 The Tuple Relational Calculus

When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query. The tuple relational calculus, by contrast, is a **nonprocedural** query language. It describes the desired information without giving a specific procedure for obtaining that information.

A query in the tuple relational calculus is expressed as:

$$\{t \mid P(t)\}$$

That is, it is the set of all tuples t such that predicate P is true for t. Following our earlier notation, we use $t[A]$ to denote the value of tuple t on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.

Before we give a formal definition of the tuple relational calculus, we return to some of the queries for which we wrote relational-algebra expressions in Section 6.1.1.

6.2.1 Example Queries

Find the *ID*, *name*, *dept_name*, *salary* for instructors whose salary is greater than $80,000:

$$\{t \mid t \in instructor \wedge t[salary] > 80000\}$$

Suppose that we want only the *ID* attribute, rather than all attributes of the *instructor* relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (*ID*). We need those tuples on (*ID*) such that there is a tuple in *instructor* with the *salary* attribute > 80000. To express this request, we need the construct "there exists" from mathematical logic. The notation:

$$\exists t \in r\ (Q(t))$$

means "there exists a tuple t in relation r such that predicate $Q(t)$ is true."

Using this notation, we can write the query "Find the instructor *ID* for each instructor with a salary greater than $80,000" as:

$$\{t \mid \exists s \in instructor\ (t[ID] = s[ID] \wedge s[salary] > 80000)\}$$

In English, we read the preceding expression as "The set of all tuples t such that there exists a tuple s in relation *instructor* for which the values of t and s for the *ID* attribute are equal, and the value of s for the *salary* attribute is greater than $80,000."

Tuple variable t is defined on only the *ID* attribute, since that is the only attribute having a condition specified for t. Thus, the result is a relation on (*ID*).

Consider the query "Find the names of all instructors whose department is in the Watson building." This query is slightly more complex than the previous queries, since it involves two relations: *instructor* and *department*. As we shall see, however, all it requires is that we have two "there exists" clauses in our tuple-relational-calculus expression, connected by *and* ($\wedge$). We write the query as follows:

$$\{t \mid \exists s \in instructor\ (t[name] = s[name] \wedge \exists u \in department\ (u[dept\_name] = s[dept\_name] \wedge u[building] = \text{"Watson"}))\}$$

name
Einstein
Crick
Gold

Figure 6.21 Names of all instructors whose department is in the Watson building.

Tuple variable u is restricted to departments that are located in the Watson building, while tuple variable s is restricted to instructors whose *dept_name* matches that of tuple variable u. Figure 6.21 shows the result of this query.

To find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both, we used the union operation in the relational algebra. In the tuple relational calculus, we shall need two "there exists" clauses, connected by *or* ($\vee$):

$$\{t \mid \exists s \in section\ (t[course\_id] = s[course\_id] \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009) \vee \exists u \in section\ (u[course\_id] = t[course\_id]) \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$$

This expression gives us the set of all *course_id* tuples for which at least one of the following holds:

- The *course_id* appears in some tuple of the *section* relation with *semester* = Fall and *year* = 2009.
- The *course_id* appears in some tuple of the *section* relation with *semester* = Spring and *year* = 2010.

If the same course is offered in both the Fall 2009 and Spring 2010 semesters, its *course_id* appears only once in the result, because the mathematical definition of a set does not allow duplicate members. The result of this query appeared earlier in Figure 6.5.

If we now want *only* those *course_id* values for courses that are offered in *both* the Fall 2009 and Spring 2010 semesters, all we need to do is to change the *or* ($\vee$) to *and* ($\wedge$) in the preceding expression.

$$\{t \mid \exists s \in section\ (t[course\_id] = s[course\_id] \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009) \wedge \exists u \in section\ (u[course\_id] = t[course\_id]) \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$$

The result of this query appeared in Figure 6.13.

Now consider the query "Find all the courses taught in the Fall 2009 semester but not in Spring 2010 semester." The tuple-relational-calculus expression for this

query is similar to the expressions that we have just seen, except for the use of the *not* ($\neg$) symbol:

$$\{t \mid \exists s \in section\ (t[course\_id] = s[course\_id]) \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009)\} \wedge \neg \exists u \in section\ (u[course\_id] = t[course\_id]) \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$$

This tuple-relational-calculus expression uses the $\exists s \in section\ (\ldots)$ clause to require that a particular *course_id* is taught in the Fall 2009 semester, and it uses the $\neg \exists u \in section\ (\ldots)$ clause to eliminate those *course_id* values that appear in some tuple of the *section* relation as having been taught in the Spring 2010 semester.

The query that we shall consider next uses implication, denoted by $\Rightarrow$. The formula $P \Rightarrow Q$ means "P implies Q"; that is, "if P is true, then Q must be true." Note that $P \Rightarrow Q$ is logically equivalent to $\neg P \vee Q$. The use of implication rather than *not* and *or* often suggests a more intuitive interpretation of a query in English.

Consider the query that "Find all students who have taken all courses offered in the Biology department." To write this query in the tuple relational calculus, we introduce the "for all" construct, denoted by $\forall$. The notation:

$$\forall t \in r\ (Q(t))$$

means "Q is true for all tuples t in relation r."

We write the expression for our query as follows:

$$\{t \mid \exists r \in student\ (r[ID] = t[ID]) \wedge (\forall u \in course\ (u[dept\_name] = \text{"Biology"} \Rightarrow \exists s \in takes\ (t[ID] = s[ID] \wedge s[course\_id] = u[course\_id]))\}$$

In English, we interpret this expression as "The set of all students (that is, (*ID*) tuples t) such that, for *all* tuples u in the *course* relation, if the value of u on attribute *dept_name* is 'Biology', then there exists a tuple in the *takes* relation that includes the student *ID* and the *course_id*."

Note that there is a subtlety in the above query: If there is no course offered in the Biology department, all student *ID*s satisfy the condition. The first line of the query expression is critical in this case—without the condition

$$\exists r \in student\ (r[ID] = t[ID])$$

if there is no course offered in the Biology department, any value of t (including values that are not student *ID*s in the *student* relation) would qualify.

6.2.2 Formal Definition

We are now ready for a formal definition. A tuple-relational-calculus expression is of the form:

$$\{t \mid P(t)\}$$

where P is a *formula*. Several tuple variables may appear in a formula. A tuple variable is said to be a *free variable* unless it is quantified by a $\exists$ or $\forall$. Thus, in:

$$t \in instructor \wedge \exists s \in department(t[dept\_name] = s[dept\_name])$$

t is a free variable. Tuple variable s is said to be a *bound* variable.

A tuple-relational-calculus formula is built up out of *atoms*. An atom has one of the following forms:

- $s \in r$, where s is a tuple variable and r is a relation (we do not allow use of the $\notin$ operator).
- $s[x]\ \Theta\ u[y]$, where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison operator ($<, \leq, =, \neq, >, \geq$); we require that attributes x and y have domains whose members can be compared by Θ.
- $s[x]\ \Theta\ c$, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.
- If $P_1(s)$ is a formula containing a free tuple variable s, and r is a relation, then

$$\exists s \in r\ (P_1(s)) \text{ and } \forall s \in r\ (P_1(s))$$

are also formulae.

As we could for the relational algebra, we can write equivalent expressions that are not identical in appearance. In the tuple relational calculus, these equivalences include the following three rules:

1. $P_1 \wedge P_2$ is equivalent to $\neg\ (\neg(P_1) \vee \neg(P_2))$.
2. $\forall t \in r\ (P_1(t))$ is equivalent to $\neg \exists t \in r\ (\neg P_1(t))$.
3. $P_1 \Rightarrow P_2$ is equivalent to $\neg(P_1) \vee P_2$.

6.2.3 Safety of Expressions

There is one final issue to be addressed. A tuple-relational-calculus expression may generate an infinite relation. Suppose that we write the expression:

$$\{t \mid \neg (t \in instructor)\}$$

There are infinitely many tuples that are not in *instructor*. Most of these tuples contain values that do not even appear in the database! Clearly, we do not wish to allow such expressions.

To help us define a restriction of the tuple relational calculus, we introduce the concept of the **domain** of a tuple relational formula, P. Intuitively, the domain of P, denoted $dom(P)$, is the set of all values referenced by P. They include values mentioned in P itself, as well as values that appear in a tuple of a relation mentioned in P. Thus, the domain of P is the set of all values that appear explicitly in P or that appear in one or more relations whose names appear in P. For example, $dom(t \in instructor \wedge t[salary] > 80000)$ is the set containing 80000 as well as the set of all values appearing in any attribute of any tuple in the *instructor* relation. Similarly, $dom(\neg (t \in instructor))$ is also the set of all values appearing in *instructor*, since the relation *instructor* is mentioned in the expression.

We say that an expression $\{t \mid P(t)\}$ is *safe* if all values that appear in the result are values from $dom(P)$. The expression $\{t \mid \neg (t \in instructor)\}$ is not safe. Note that $dom(\neg (t \in instructor))$ is the set of all values appearing in *instructor*. However, it is possible to have a tuple t not in *instructor* that contains values that do not appear in *instructor*. The other examples of tuple-relational-calculus expressions that we have written in this section are safe.

The number of tuples that satisfy an unsafe expression, such as $\{t \mid \neg (t \in instructor)\}$, could be infinite, whereas safe expressions are guaranteed to have finite results. The class of tuple-relational-calculus expressions that are allowed is therefore restricted to those that are safe.

6.2.4 Expressive Power of Languages

The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the basic relational algebra (with the operators $\cup$, $-$, $\times$, σ, and ρ, but without the extended relational operations such as generalized projection and aggregation ($\mathcal{G}$)). Thus, for every relational-algebra expression using only the basic operations, there is an equivalent expression in the tuple relational calculus, and for every tuple-relational-calculus expression, there is an equivalent relational-algebra expression. We shall not prove this assertion here; the bibliographic notes contain references to the proof. Some parts of the proof are included in the exercises. We note that the tuple relational calculus does not have any equivalent of the aggregate operation, but it can be extended to support aggregation. Extending the tuple relational calculus to handle arithmetic expressions is straightforward.

6.3 The Domain Relational Calculus

A second form of relational calculus, called **domain relational calculus**, uses *domain* variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.

Domain relational calculus serves as the theoretical basis of the widely used QBE language (see Appendix B.1), just as relational algebra serves as the basis for the SQL language.

6.3.1 Formal Definition

An expression in the domain relational calculus is of the form

$$\{< x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

where $x_1, x_2, \ldots, x_n$ represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

- $< x_1, x_2, \ldots, x_n > \in r$, where r is a relation on n attributes and $x_1, x_2, \ldots, x_n$ are domain variables or domain constants.
- $x \Theta y$, where x and y are domain variables and Θ is a comparison operator ($<, \leq, =, \neq, >, \geq$). We require that attributes x and y have domains that can be compared by Θ.
- $x \Theta c$, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.
- If $P_1(x)$ is a formula in x, where x is a free domain variable, then

  $$\exists\, x\, (P_1(x)) \text{ and } \forall\, x\, (P_1(x))$$

  are also formulae.

As a notational shorthand, we write $\exists a, b, c\, (P(a, b, c))$ for $\exists a\, (\exists b\, (\exists c\, (P(a, b, c))))$.

6.3.2 Example Queries

We now give domain-relational-calculus queries for the examples that we considered earlier. Note the similarity of these expressions and the corresponding tuple-relational-calculus expressions.

- Find the instructor *ID*, *name*, *dept_name*, and *salary* for instructors whose salary is greater than \$80,000:

$$\{< i, n, d, s > \mid < i, n, d, s > \in \textit{instructor} \wedge s > 80000\}$$

- Find all instructor *ID* for instructors whose salary is greater than \$80,000:

$$\{< n > \mid \exists\, i, d, s\, (< i, n, d, s > \in \textit{instructor} \wedge s > 80000)\}$$

Although the second query appears similar to the one that we wrote for the tuple relational calculus, there is an important difference. In the tuple calculus, when we write $\exists\, s$ for some tuple variable s, we bind it immediately to a relation by writing $\exists\, s \in r$. However, when we write $\exists\, n$ in the domain calculus, n refers not to a tuple, but rather to a domain value. Thus, the domain of variable n is unconstrained until the subformula $< i, n, d, s > \in \textit{instructor}$ constrains n to instructor names that appear in the *instructor* relation.

We now give several examples of queries in the domain relational calculus.

- Find the names of all instructors in the Physics department together with the *course_id* of all courses they teach:

$$\begin{aligned}\{< n, c > \mid \exists\, i, a\, (< i, c, a, s, y > \in \textit{teaches} \\ \wedge \exists\, d, s\, (< i, n, d, s > \in \textit{instructor} \wedge d = \text{“Physics”}))\}\end{aligned}$$

- Find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both:

$$\begin{aligned}\{< c > \mid \exists s\, (< c, a, s, y, b, r, t > \in \textit{section} \\ \wedge s = \text{“Fall”} \wedge y = \text{“2009”} \\ \vee \exists u\, (< c, a, s, y, b, r, t > \in \textit{section} \\ \wedge s = \text{“Spring”} \wedge y = \text{“2010”}\end{aligned}$$

- Find all students who have taken all courses offered in the Biology department:

$$\begin{aligned}\{< i > \mid \exists\, n,\ d, t\, (< i, n, d, t > \in \textit{student}) \wedge \\ \forall\, x, y, z, w\, (< x, y, z, w > \in \textit{course} \wedge z = \text{“Biology”} \Rightarrow \\ \exists\, a, b\, (< a, x, b, r, p, q > \in \textit{takes} \wedge < c, a > \in \textit{depositor}))\}\end{aligned}$$

Note that as was the case for tuple-relational-calculus, if no courses are offered in the Biology department, all students would be in the result.

6.3.3 Safety of Expressions

We noted that, in the tuple relational calculus (Section 6.2), it is possible to write expressions that may generate an infinite relation. That led us to define *safety* for tuple-relational-calculus expressions. A similar situation arises for the domain relational calculus. An expression such as

$$\{< i, n, d, s > \mid \neg(< i, n, d, s > \in \textit{instructor})\}$$

is unsafe, because it allows values in the result that are not in the domain of the expression.

For the domain relational calculus, we must be concerned also about the form of formulae within “there exists” and “for all” clauses. Consider the expression

$$\{< x > \mid \exists\, y\, (< x,\ y > \in r) \wedge \exists\, z\, (\neg(< x,\ z > \in r) \wedge P(x, z))\}$$

where P is some formula involving x and z. We can test the first part of the formula, $\exists\, y\, (< x,\ y > \in r)$, by considering only the values in r. However, to test the second part of the formula, $\exists\, z\, (\neg\, (< x,\ z > \in r) \wedge P(x, z))$, we must consider values for z that do not appear in r. Since all relations are finite, an infinite number of values do not appear in r. Thus, it is not possible, in general, to test the second part of the formula without considering an infinite number of potential values for z. Instead, we add restrictions to prohibit expressions such as the preceding one.

In the tuple relational calculus, we restricted any existentially quantified variable to range over a specific relation. Since we did not do so in the domain calculus, we add rules to the definition of safety to deal with cases like our example. We say that an expression

$$\{< x_1,\ x_2, \ldots, x_n > \mid P\,(x_1,\ x_2, \ldots, x_n)\}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from $dom(P)$.
2. For every “there exists” subformula of the form $\exists\, x\ (P_1(x))$, the subformula is true if and only if there is a value x in $dom(P_1)$ such that $P_1(x)$ is true.
3. For every “for all” subformula of the form $\forall x\ (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from $dom(P_1)$.

The purpose of the additional rules is to ensure that we can test “for all” and “there exists” subformulae without having to test infinitely many possibilities. Consider the second rule in the definition of safety. For $\exists\ x\ (P_1(x))$ to be true,

we need to find only one x for which $P_1(x)$ is true. In general, there would be infinitely many values to test. However, if the expression is safe, we know that we can restrict our attention to values from $dom(P_1)$. This restriction reduces to a finite number the tuples we must consider.

The situation for subformulae of the form $\forall x\ (P_1(x))$ is similar. To assert that $\forall x\ (P_1(x))$ is true, we must, in general, test all possible values, so we must examine infinitely many values. As before, if we know that the expression is safe, it is sufficient for us to test $P_1(x)$ for those values taken from $dom(P_1)$.

All the domain-relational-calculus expressions that we have written in the example queries of this section are safe, except for the example unsafe query we saw earlier.

6.3.4 Expressive Power of Languages

When the domain relational calculus is restricted to safe expressions, it is equivalent in expressive power to the tuple relational calculus restricted to safe expressions. Since we noted earlier that the restricted tuple relational calculus is equivalent to the relational algebra, all three of the following are equivalent:

- The basic relational algebra (without the extended relational-algebra operations)
- The tuple relational calculus restricted to safe expressions
- The domain relational calculus restricted to safe expressions

We note that the domain relational calculus also does not have any equivalent of the aggregate operation, but it can be extended to support aggregation, and extending it to handle arithmetic expressions is straightforward.

6.4 Summary

- The **relational algebra** defines a set of algebraic operations that operate on tables, and output tables as their results. These operations can be combined to get expressions that express desired queries. The algebra defines the basic operations used within relational query languages.
- The operations in relational algebra can be divided into:
  - Basic operations
  - Additional operations that can be expressed in terms of the basic operations
  - Extended operations, some of which add further expressive power to relational algebra
- The relational algebra is a terse, formal language that is inappropriate for casual users of a database system. Commercial database systems, therefore, use languages with more "syntactic sugar." In Chapters 3 through 5, we cover the most influential language—SQL, which is based on relational algebra.
- The **tuple relational calculus** and the **domain relational calculus** are nonprocedural languages that represent the basic power required in a relational query language. The basic relational algebra is a procedural language that is equivalent in power to both forms of the relational calculus when they are restricted to safe expressions.
- The relational calculi are terse, formal languages that are inappropriate for casual users of a database system. These two formal languages form the basis for two more user-friendly languages, QBE and Datalog, that we cover in Appendix B.

Review Terms

- Relational algebra
- Relational-algebra operations
  - Select σ
  - Project Π
  - Union $\cup$
  - Set difference $-$
  - Cartesian product $\times$
  - Rename ρ
- Additional operations
  - Set intersection $\cap$
  - Natural join $\bowtie$
  - Assignment operation
  - Outer join
    - Left outer join $\mathbin{⟕}$
    - Right outer join $\mathbin{⟖}$
    - Full outer join $\mathbin{⟗}$
- Multisets
- Grouping
- Null value
- Tuple relational calculus
- Domain relational calculus
- Safety of expressions
- Expressive power of languages

Practice Exercises

6.1 Write the following queries in relational algebra, using the university schema.

a. Find the titles of courses in the Comp. Sci. department that have 3 credits.

b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.

c. Find the highest salary of any instructor.

d. Find all instructors earning the highest salary (there may be more than one with the same salary).

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*person_name*, *manager_name*)

Figure 6.22 Relational database for Exercises 6.2, 6.8, 6.11, 6.13, and 6.15

e. Find the enrollment of each section that was offered in Autumn 2009.

f. Find the maximum enrollment, across all sections, in Autumn 2009.

g. Find the sections that had the maximum enrollment in Autumn 2009.

6.2 Consider the relational database of Figure 6.22, where the primary keys are underlined. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who live in the same city and on the same street as do their managers.

b. Find the names of all employees in this database who do not work for "First Bank Corporation".

c. Find the names of all employees who earn more than every employee of "Small Bank Corporation".

6.3 The natural outer-join operations extend the natural-join operation so that tuples from the participating relations are not lost in the result of the join. Describe how the theta-join operation can be extended so that tuples from the left, right, or both relations are not lost from the result of a theta join.

6.4 **(Division operation)**: The division operator of relational algebra, "÷", is defined as follows. Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$; that is, every attribute of schema S is also in schema R. Then $r \div s$ is a relation on schema $R - S$ (that is, on the schema containing all attributes of schema R that are not in schema S). A tuple t is in $r \div s$ if and only if both of two conditions hold:

- t is in $\Pi_{R-S}(r)$
- For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:
  a. $t_r[S] = t_s[S]$
  b. $t_r[R - S] = t$

Given the above definition:

a. Write a relational algebra expression using the division operator to find the IDs of all students who have taken all Comp. Sci. courses. (Hint: project *takes* to just ID and *course_id*, and generate the set of all Comp. Sci. *course_id*s using a select expression, before doing the division.)

b. Show how to write the above query in relational algebra, without using division. (By doing so, you would have shown how to define the division operation using the other relational algebra operations.)

6.5 Let the following relation schemas be given:

$$R = (A, B, C)$$
$$S = (D, E, F)$$

Let relations $r(R)$ and $s(S)$ be given. Give an expression in the tuple relational calculus that is equivalent to each of the following:

a. $\Pi_A(r)$

b. $\sigma_{B=17}(r)$

c. $r \times s$

d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

6.6 Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R. Give an expression in the domain relational calculus that is equivalent to each of the following:

a. $\Pi_A(r_1)$

b. $\sigma_{B=17}(r_1)$

c. $r_1 \cup r_2$

d. $r_1 \cap r_2$

e. $r_1 - r_2$

f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

6.7 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write expressions in relational algebra for each of the following queries:

a. $\{< a > \mid \exists b\,(< a, b > \in r \wedge b = 7)\}$

b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$

c. $\{< a > \mid \exists c\,(< a, c > \in s \wedge \exists b_1, b_2\,(< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

6.8 Consider the relational database of Figure 6.22 where the primary keys are underlined. Give an expression in tuple relational calculus for each of the following queries:

a. Find all employees who work directly for "Jones."

b. Find all cities of residence of all employees who work directly for "Jones."

c. Find the name of the manager of the manager of "Jones."

d. Find those employees who earn more than all employees living in the city "Mumbai."

6.9 Describe how to translate join expressions in SQL to relational algebra.

Exercises

6.10 Write the following queries in relational algebra, using the university schema.

a. Find the names of all students who have taken at least one Comp. Sci. course.

b. Find the IDs and names of all students who have not taken any course offering before Spring 2009.

c. For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.

d. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.

6.11 Consider the relational database of Figure 6.22, where the primary keys are underlined. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who work for "First Bank Corporation".

b. Find the names and cities of residence of all employees who work for "First Bank Corporation".

c. Find the names, street addresses, and cities of residence of all employees who work for "First Bank Corporation" and earn more than $10,000.

d. Find the names of all employees in this database who live in the same city as the company for which they work.

e. Assume the companies may be located in several cities. Find all companies located in every city in which "Small Bank Corporation" is located.

6.12 Using the university example, write relational-algebra queries to find the course sections taught by more than one instructor in the following ways:

a. Using an aggregate function.

b. Without using any aggregate functions.

6.13 Consider the relational database of Figure 6.22. Give a relational-algebra expression for each of the following queries:

a. Find the company with the most employees.

b. Find the company with the smallest payroll.

c. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

6.14 Consider the following relational schema for a library:

member(*memb_no*, *name*, *dob*)
books(*isbn*, *title*, *authors*, *publisher*)
borrowed(*memb_no*, *isbn*, *date*)

Write the following queries in relational algebra.

a. Find the names of members who have borrowed any book published by "McGraw-Hill".

b. Find the name of members who have borrowed all books published by "McGraw-Hill".

c. Find the name and membership number of members who have borrowed more than five different books published by "McGraw-Hill".

d. For each publisher, find the name and membership number of members who have borrowed more than five books of that publisher.

e. Find the average number of books borrowed per member. Take into account that if an member does not borrow any books, then that member does not appear in the *borrowed* relation at all.

6.15 Consider the employee database of Figure 6.22. Give expressions in tuple relational calculus and domain relational calculus for each of the following queries:

a. Find the names of all employees who work for "First Bank Corporation".

b. Find the names and cities of residence of all employees who work for "First Bank Corporation".

c. Find the names, street addresses, and cities of residence of all employees who work for "First Bank Corporation" and earn more than $10,000.

d. Find all employees who live in the same city as that in which the company for which they work is located.

e. Find all employees who live in the same city and on the same street as their managers.

f. Find all employees in the database who do not work for "First Bank Corporation".

g. Find all employees who earn more than every employee of "Small Bank Corporation".

h. Assume that the companies may be located in several cities. Find all companies located in every city in which "Small Bank Corporation" is located.

6.16 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write relational-algebra expressions equivalent to the following domain-relational-calculus expressions:

a. $\{< a > \mid \exists b\, (< a, b > \in r \wedge b = 17)\}$

b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$

c. $\{< a > \mid \exists b\, (< a, b > \in r) \vee \forall c\, (\exists d\, (< d, c > \in s) \Rightarrow < a, c > \in s)\}$

d. $\{< a > \mid \exists c\, (< a, c > \in s \wedge \exists b_1, b_2\, (< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

6.17 Repeat Exercise 6.16, writing SQL queries instead of relational-algebra expressions.

6.18 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Using the special constant *null*, write tuple-relational-calculus expressions equivalent to each of the following:

a. $r \ltimes s$

b. $r \overline{\ltimes} s$

c. $r \overline{\bowtie} s$

6.19 Give a tuple-relational-calculus expression to find the maximum value in relation $r(A)$.

Bibliographical Notes

The original definition of relational algebra is in Codd [1970]. Extensions to the relational model and discussions of incorporation of null values in the relational algebra (the RM/T model), as well as outer joins, are in Codd [1979]. Codd [1990] is a compendium of E. F. Codd's papers on the relational model. Outer joins are also discussed in Date [1993b].

The original definition of tuple relational calculus is in Codd [1972]. A formal proof of the equivalence of tuple relational calculus and relational algebra is in Codd [1972]. Several extensions to the relational calculus have been proposed. Klug [1982] and Escobar-Molano et al. [1993] describe extensions to scalar aggregate functions.