CHAPTER 27

Formal-Relational Query Languages

In Chapter 2 we introduced the relational model and presented the relational algebra (RA), which forms the basis of the widely used SQL query language. In this chapter we continue with our coverage of “pure” query languages. In particular, we cover the tuple relational calculus and the domain relational calculus, which are declarative query languages based on mathematical logic. We also cover Datalog, which has a syntax modeled after the Prolog language. Although not used commercially at present, Datalog has been used in several research database systems. For Datalog, we present fundamental constructs and concepts rather than a complete users’ guide for these languages. Keep in mind that individual implementations of a language may differ in details or may support only a subset of the full language.

27.1 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 2.6.

27.1.1 Example Queries

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

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

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:

$$\begin{aligned}\{t \mid \exists\, s \in instructor\,(t[ID] &= s[ID]\\ &\land s[salary] > 80000)\}\end{aligned}$$

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* ($\land$). We write the query as follows:

$$\begin{aligned}\{t \mid \exists\, s \in instructor\,(t[name] &= s[name]\\ \land \exists\, u \in department\,(u[dept\_name] &= s[dept\_name]\\ \land\, u[building] &= \text{“Watson”}))\}\end{aligned}$$

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 27.1 shows the result of this query.

To find the set of all courses taught in the Fall 2017 semester, the Spring 2018 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* ($\lor$):

name
Einstein
Crick
Gold

Figure 27.1 Names of all instructors whose department is in the Watson building.

$$\begin{aligned}\{t \mid \exists s \in \textit{section}\ (t[\textit{course\_id}] &= s[\textit{course\_id}]) \\ \wedge\ s[\textit{semester}] = \text{“Fall”} \wedge s[\textit{year}] &= 2017) \\ \vee\ \exists u \in \textit{section}\ (u[\textit{course\_id}] &= t[\textit{course\_id}]) \\ \wedge\ u[\textit{semester}] = \text{“Spring”} \wedge u[\textit{year}] &= 2018)\}\end{aligned}$$

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* = 2017.
- The *course_id* appears in some tuple of the *section* relation with *semester* = Spring and *year* = 2018.

If the same course is offered in both the Fall 2017 and Spring 2018 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 is shown in Figure 27.2.

If we now want *only* those *course_id* values for courses that are offered in *both* the Fall 2017 and Spring 2018 semesters, all we need to do is to change the *or* ($\vee$) to *and* ($\wedge$) in the preceding expression.

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

Figure 27.2 Courses offered in either Fall 2017, Spring 2018, or both semesters.

course_id
CS-101

Figure 27.3 Courses offered in both the Fall 2017 and Spring 2018 semesters.

$$\begin{aligned}\{t \mid \exists\, s \in section\ (t[course\_id] &= s[course\_id])\\ &\land s[semester] = \text{“Fall”} \land s[year] = 2017)\\ \land \exists\, u \in section\ (u[course\_id] &= t[course\_id])\\ &\land u[semester] = \text{“Spring”} \land u[year] = 2018)\}\end{aligned}$$

The result of this query appears in Figure 27.3.

Now consider the query “Find all the courses taught in the Fall 2017 semester but not in Spring 2018 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:

$$\begin{aligned}\{t \mid \exists\, s \in section\ (t[course\_id] &= s[course\_id])\\ &\land s[semester] = \text{“Fall”} \land s[year] = 2017)\\ \land \neg \exists\, u \in section\ (u[course\_id] &= t[course\_id])\\ &\land u[semester] = \text{“Spring”} \land u[year] = 2018)\}\end{aligned}$$

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 2017 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 2018 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 $\neg P \lor Q$. 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:

$$\begin{aligned}\{t \mid \exists\, r \in student\ (r[ID] = t[ID]) \land\\ (\forall\, u \in course\ (u[dept\_name] = \text{“ Biology”} \Rightarrow\\ \exists\, s \in takes\ (t[ID] = s[ID]\\ \land s[course\_id] = u[course\_id]))\}\end{aligned}$$

In English, we interpret this expression as "The set of all students (i.e., (*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 preceding 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

$$\exists\, r \in \mathit{student}\ (r[ID] = t[ID])$$

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.

27.1.2 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:

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

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 $P_1 \lor P_2$, $P_1 \land P_2$, 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. $P_1 \land P_2$ is equivalent to $\neg\,(\neg(P_1) \lor \neg(P_2))$.
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 $\neg(P_1) \lor P_2$.

27.1.3 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:

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

There are infinitely many tuples that are not in *instructor*. Most of these tuples contain values that do not even appear in the database! 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, $dom(t \in \mathit{instructor} \land t[\mathit{salary}] > 80000)$ 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, $dom(\neg\,(t \in \mathit{instructor}))$ 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 $\{t \mid \neg\,(t \in \mathit{instructor})\}$ is not safe. Note that $dom(\neg\,(t \in \mathit{instructor}))$ 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.

27.2 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 QBE language just as relational algebra serves as the basis for the SQL language.

27.2.1 Formal Definition

An expression in the domain relational calculus is of the form

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

where $x_1,\ x_2, \dots , x_n$ 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:

- $< x_1,\ x_2, \dots , x_n > \in\ r$, where r is a relation on n attributes and $x_1,\ x_2, \dots , x_n$ 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 $\exists\, a\ (\exists\, b\ (\exists\, c\ (P(a, b, c))))$.

27.2.2 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:

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

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

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

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 $< i, n, d, s > \in\ instructor$ 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:

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

- Find the set of all courses taught in the Fall 2017 semester, the Spring 2018 semester, or both:

$$\begin{aligned}\{ < c > \ | \ \exists\, a, s, y, b, r, t\ (&< c, a, s, y, b, r, t > \in\ section \\ &\wedge s\ =\ \text{“Fall”} \wedge y\ =\ \text{“2017”}) \\ \vee \exists\, a, s, y, b, r, t\ (&< c, a, s, y, b, r, t > \in\ section \\ &\wedge s\ =\ \text{“Spring”} \wedge y\ =\ \text{“2018”})\}\end{aligned}$$

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

$$\begin{aligned}\{ < i > \mid & \exists\, n,\ d, tc\ (< i, n, d, tc > \in\ student) \land \\ & \forall\, ci, ti, dn, cr\ (< ci, ti, dn, cr > \in\ course\ \land\ dn\ =\ \text{“Biology”}\ \Rightarrow \\ & \exists\, si, se, y, g\ (< i, ci, si, se, y, g > \in\ takes\ ))\}\end{aligned}$$

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.

27.2.3 Safety of Expressions

We noted that, in the tuple relational calculus (Section 27.1), 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

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

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

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

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, $\exists\, z\ (\neg\ (< x,\ z > \in\ r)\ \land\ P(x, z))$, 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.

27.3 Expressive Power of Pure Relational Query 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, Π, and ρ, but without the extended relational operations such as generalized projection and aggregation (γ)). 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.

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.

account_number	*branch_name*	*balance*
A-101	Downtown	500
A-215	Minus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Perryridge	900
A-222	Redwood	700
A-217	Perryridge	750

Figure 27.4 The *account* relation.

27.4 Datalog

Datalog is a nonprocedural query language based on the logic-programming language Prolog. As in the relational calculus, a user describes the information desired without giving a specific procedure for obtaining that information. The syntax of Datalog resembles that of Prolog. However, the meaning of Datalog programs is defined in a purely declarative manner, unlike the more procedural semantics of Prolog, so Datalog simplifies writing simple queries and makes query optimization easier.

27.4.1 Basic Structure

A Datalog program consists of a set of **rules**. Before presenting a formal definition of Datalog rules and their formal meaning, we consider examples. Consider a Datalog rule to define a view relation *v1* containing account numbers and balances for accounts at the Perryridge branch with a balance of over $700:

$$v1(A, B) \ :\!-\ account(A, \text{“Perryridge”}, B),\ B > 700$$

Datalog rules define views; the preceding rule **uses** the relation *account*, and **defines** the view relation *v1*. The symbol :– is read as “if,” and the comma separating the “*account*(*A*, “Perryridge”, *B*)” from “*B* > 700” is read as “and.” Intuitively, the rule is understood as follows:

$$\begin{array}{ll} \textbf{for all} & A, B \\ \textbf{if} & (A, \text{“Perryridge”}, B) \in account \ \textbf{and}\ B > 700 \\ \textbf{then} & (A, B) \in v1 \end{array}$$

Suppose that the relation *account* is as shown in Figure 27.4. Then, the view relation *v1* contains the tuples in Figure 27.5.

To retrieve the balance of account number A-217 in the view relation *v1*, we can write the following query:

account_number	*balance*
A-201	900
A-217	750

Figure 27.5 The *v1* relation.

$$? \ v1(\text{"A-217"}, B)$$

The answer to the query is

$$(\text{A-217}, 750)$$

To get the account number and balance of all accounts in relation *v1*, where the balance is greater than 800, we can write

$$? \ v1(A, B), B > 800$$

The answer to this query is

$$(\text{A-201}, 900)$$

In general, we need more than one rule to define a view relation. Each rule defines a set of tuples that the view relation must contain. The set of tuples in the view relation is then defined as the union of all these sets of tuples. The following Datalog program specifies the interest rates for accounts:

$$\begin{aligned} &interest\_rate(A, 5) \ :\!-\ account(A, N, B), B < 10000 \\ &interest\_rate(A, 6) \ :\!-\ account(A, N, B), B >= 10000 \end{aligned}$$

The program has two rules defining a view relation *interest_rate*, whose attributes are the account number and the interest rate. The rules say that, if the balance is less than $10,000, then the interest rate is 5 percent, and if the balance is greater than or equal to $10,000, the interest rate is 6 percent.

Datalog rules can also use negation. The following rules define a view relation *c* that contains the names of all customers who have a deposit, but have no loan, at the bank:

$$\begin{aligned} &c(N) \ :\!-\ depositor(N,A), \textbf{not } is\_borrower(N) \\ &is\_borrower(N) \ :\!-\ borrower(N, L) \end{aligned}$$

Prolog and most Datalog implementations recognize attributes of a relation by position and omit attribute names. Thus, Datalog rules are compact, compared to SQL queries. However, when relations have a large number of attributes, or the order or number of attributes of relations may change, the positional notation can be cumbersome and error prone. It is not hard to create a variant of Datalog syntax using named attributes, rather than positional attributes. In such a system, the Datalog rule defining *v1* can be written as

> *v1*(*account_number A*, *balance B*) :-
> *account*(*account_number A*, *branch_name* “Perryridge”, *balance B*),
> $B > 700$

Translation between the two forms can be done without significant effort, given the relation schema.

27.4.2 Syntax of Datalog Rules

Now that we have informally explained rules and queries, we can formally define their syntax; we discuss their meaning in Section 27.4.3. We use the same conventions as in the relational algebra for denoting relation names, attribute names, and constants (such as numbers or quoted strings). We use uppercase (capital) letters and words starting with uppercase letters to denote variable names, and lowercase letters and words starting with lowercase letters to denote relation names and attribute names. Examples of constants are 4, which is a number, and “John,” which is a string; *X* and *Name* are variables. A **positive literal** has the form

$$p(t_1, t_2, \ldots, t_n)$$

where p is the name of a relation with n attributes, and $t_1, t_2, \ldots, t_n$ are either constants or variables. A **negative literal** has the form

$$\textbf{not } p(t_1, t_2, \ldots, t_n)$$

where relation p has n attributes. Here is an example of a literal:

$$account(A, \text{“Perryridge”}, B)$$

Literals involving arithmetic operations are treated specially. For example, the literal $B > 700$, although not in the syntax just described, can be conceptually understood to stand for $>(B, 700)$, which *is* in the required syntax, and where $>$ is a relation.

But what does this notation mean for arithmetic operations such as “$>$”? The relation $>$ (conceptually) contains tuples of the form (x, y) for every possible pair of values x, y such that $x > y$. Thus, (2, 1) and (5, −33) are both tuples in $>$. The (conceptual) relation $>$ is infinite. Other arithmetic operations (such as $>$, $=$, $+$, and $-$) are also treated conceptually as relations. For example, A = B + C stands conceptually for +(B, C, A), where the relation + contains every tuple (x, y, z) such that $z = x + y$.

$$\begin{aligned}
&\textit{interest}(A, I) \text{ :- } \textit{account}(A, \text{“Perryridge”}, B),\\
&\qquad\qquad\qquad \textit{interest\_rate}(A, R), I = B * R/100\\
&\textit{interest\_rate}(A, 5) \text{ :- } \textit{account}(A, N, B), B < 10000\\
&\textit{interest\_rate}(A, 6) \text{ :- } \textit{account}(A, N, B), B >= 10000
\end{aligned}$$

Figure 27.6 Datalog program that defines interest on Perryridge accounts.

A **fact** is written in the form

$$p(v_1, v_2, \dots, v_n)$$

and denotes that the tuple $(v_1, v_2, \dots, v_n)$ is in relation p. A set of facts for a relation can also be written in the usual tabular notation. A set of facts for the relations in a database schema is equivalent to an instance of the database schema. **Rules** are built out of literals and have the form

$$p(t_1, t_2, \dots, t_n) \text{ :- } L_1, L_2, \dots, L_n$$

where each L_i is a (positive or negative) literal. The literal $p(t_1, t_2, \dots, t_n)$ is referred to as the **head** of the rule, and the rest of the literals in the rule constitute the **body** of the rule.

A **Datalog program** consists of a set of rules; the order in which the rules are written has no significance. As mentioned earlier, there may be several rules defining a relation.

Figure 27.6 shows a Datalog program that defines the interest on each account in the Perryridge branch. The first rule of the program defines a view relation *interest*, whose attributes are the account number and the interest earned on the account. It uses the relation *account* and the view relation *interest_rate*. The last two rules of the program are rules that we saw earlier.

A view relation v_1 is said to **depend directly on** a view relation v_2 if v_2 is used in the expression defining v_1. In the preceding program, view relation *interest* depends directly on relations *interest_rate* and *account*. Relation *interest_rate* in turn depends directly on *account*.

A view relation v_1 is said to **depend indirectly on** view relation v_2 if there is a sequence of intermediate relations $i_1, i_2, \dots, i_n$, for some n, such that v_1 depends directly on i_1, i_1 depends directly on i_2, and so on until i_{n-1} depends on i_n.

In the example in Figure 27.6, since we have a chain of dependencies from *interest* to *interest_rate* to *account*, relation *interest* also depends indirectly on *account*.

Finally, a view relation v_1 is said to **depend on** view relation v_2 if v_1 depends either directly or indirectly on v_2.

A view relation v is said to be **recursive** if it depends on itself. A view relation that is not recursive is said to be **nonrecursive**.

$$\begin{aligned} empl(X, Y) &:\!- manager(X, Y) \\ empl(X, Y) &:\!- manager(X, Z), empl(Z, Y) \end{aligned}$$

Figure 27.7 Recursive Datalog program.

Consider the program in Figure 27.7. Here, the view relation *empl* depends on itself (because of the second rule), and is therefore recursive. In contrast, the program in Figure 27.6 is nonrecursive.

27.4.3 Semantics of Nonrecursive Datalog

We consider the formal semantics of Datalog programs. For now, we consider only programs that are nonrecursive. The semantics of recursive programs is somewhat more complicated; it is discussed in Section 27.4.6. We define the semantics of a program by starting with the semantics of a single rule.

27.4.3.1 Semantics of a Rule

A **ground instantiation of a rule** is the result of replacing each variable in the rule by some constant. If a variable occurs multiple times in a rule, all occurrences of the variable must be replaced by the same constant. Ground instantiations are often simply called **instantiations**.

Our example rule defining *v1*, and an instantiation of the rule, are:

$$\begin{aligned} &v1(A, B) \;:\!- account(A, \text{“Perryridge”}, B), B > 700 \\ &v1(\text{“A-217”}, 750) \;:\!- account(\text{“A-217”}, \text{“Perryridge”}, 750), 750 > 700 \end{aligned}$$

Here, variable A was replaced by “A-217” and variable B by 750.

A rule usually has many possible instantiations. These instantiations correspond to the various ways of assigning values to each variable in the rule.

Suppose that we are given a rule R,

$$p(t_1, t_2, \ldots, t_n) :\!- L_1, L_2, \ldots, L_n$$

and a set of facts I for the relations used in the rule (I can also be thought of as a database instance). Consider any instantiation R' of rule R:

$$p(v_1, v_2, \ldots, v_n) :\!- l_1, l_2, \ldots, l_n$$

where each literal l_i is either of the form $q_i(v_{i,1}, v_{1,2}, \ldots, v_{i,n_i})$ or of the form **not** $q_i(v_{i,1}, v_{i,2}, \ldots, v_{i,n_i})$, and where each v_i and each $v_{i,j}$ is a constant.

We say that the body of rule instantiation R' is **satisfied** in I if

1. For each positive literal $q_i(v_{i,1}, \ldots, v_{i,n_i})$ in the body of R', the set of facts I contains the fact $q(v_{i,1}, \ldots, v_{i,n_i})$.
2. For each negative literal **not** $q_j(v_{j,1}, \ldots, v_{j,n_j})$ in the body of R', the set of facts I does not contain the fact $q_j(v_{j,1}, \ldots, v_{j,n_j})$.

We define the set of facts that can be **inferred** from a given set of facts I using rule R as

$$\begin{aligned} \mathit{infer}(R, I) = \{p(t_1, \ldots, t_{n_i}) \mid & \text{ there is an instantiation } R' \text{ of } R, \\ & \text{where } p(t_1, \ldots, t_{n_i}) \text{ is the head of } R', \text{ and} \\ & \text{the body of } R' \text{ is satisfied in } I\}. \end{aligned}$$

Given a set of rules $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$, we define

$$\mathit{infer}(\mathcal{R}, I) = \mathit{infer}(R_1, I) \cup \mathit{infer}(R_2, I) \cup \ldots \cup \mathit{infer}(R_n, I)$$

Suppose that we are given a set of facts I containing the tuples for relation *account* in Figure 27.4. One possible instantiation of our running-example rule R is

$$v1(\text{“A-217”}, 750) \text{ :- } \mathit{account}(\text{“A-217”}, \text{“Perryridge”}, 750), 750 > 700$$

The fact *account*(“A-217”, “Perryridge”, 750) is in the set of facts I. Further, 750 is greater than 700, and hence conceptually (750, 700) is in the relation “>”. Hence, the body of the rule instantiation is satisfied in I. There are other possible instantiations of R, and using them we find that $\mathit{infer}(R, I)$ has exactly the set of facts for *v1* that appears in Figure 27.8.

27.4.3.2 Semantics of a Program

When a view relation is defined in terms of another view relation, the set of facts in the first view depends on the set of facts in the second one. We have assumed, in this section, that the definition is nonrecursive; that is, no view relation depends (directly or indirectly) on itself. Hence, we can layer the view relations in the following way and can use the layering to define the semantics of the program:

account_number	*balance*
A-201	900
A-217	750

Figure 27.8 Result of *infer*(*R*, *I*).

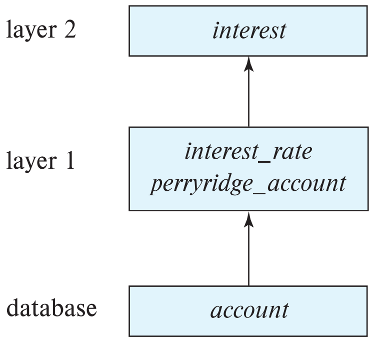

Figure 27.9 Layering of view relations.

- A relation is in layer 1 if all relations used in the bodies of rules defining it are stored in the database.
- A relation is in layer 2 if all relations used in the bodies of rules defining it either are stored in the database or are in layer 1.
- In general, a relation p is in layer $i + 1$ if (1) it is not in layers $1, 2, \ldots, i$ and (2) all relations used in the bodies of rules defining p either are stored in the database or are in layers $1, 2, \ldots, i$.

Consider the program in Figure 27.6 with the additional rule:

$$perryridge\_account(X, Y) \text{ :- } account(X, \text{“Perryridge”}, Y)$$

The layering of view relations in the program appears in Figure 27.9. The relation *account* is in the database. Relation *interest_rate* is in layer 1, since all the relations used in the two rules defining it are in the database. Relation *perryridge_account* is similarly in layer 1. Finally, relation *interest* is in layer 2, since it is not in layer 1 and all the relations used in the rule defining it are in the database or in layers lower than 2.

We can now define the semantics of a Datalog program in terms of the layering of view relations. Let the layers in a given program be $1, 2, \ldots, n$. Let $\mathcal{R}_i$ denote the set of all rules defining view relations in layer i.

- We define I_0 to be the set of facts stored in the database, and we define I_1 as

$$I_1 = I_0 \cup infer(\mathcal{R}_1, I_0)$$

- We proceed in a similar fashion, defining I_2 in terms of I_1 and $\mathcal{R}_2$, and so on, using the following definition:

$$I_{i+1} = I_i \cup infer(\mathcal{R}_{i+1}, I_i)$$

- Finally, the set of facts in the view relations defined by the program (also called the **semantics of the program**) is given by the set of facts I_n corresponding to the highest layer n.

For the program in Figure 27.6, I_0 is the set of facts in the database, and I_1 is the set of facts in the database along with all facts that we can infer from I_0 using the rules for relations *interest_rate* and *perryridge_account*. Finally, I_2 contains the facts in I_1 along with the facts for relation *interest* that we can infer from the facts in I_1 by the rule defining *interest*. The semantics of the program—that is, the set of those facts that are in each of the view relations—is defined as the set of facts I_2.

27.4.4 Safety

It is possible to write rules that generate an infinite number of answers. Consider the rule

$$gt(X, Y) \text{ :- } X > Y$$

Since the relation defining > is infinite, this rule would generate an infinite number of facts for the relation *gt*, which calculation would, correspondingly, take an infinite amount of time and space.

The use of negation can also cause similar problems. Consider the rule:

$$not\_in\_loan(L, B, A) \text{ :- } \textbf{not } loan(L, B, A)$$

The idea is that a tuple *(loan_number, branch_name, amount)* is in view relation *not_in _loan* if the tuple is not present in the *loan* relation. However, if the set of possible loan_numbers, branch_names, and balances is infinite, the relation *not_in_loan* would be infinite as well.

Finally, if we have a variable in the head that does not appear in the body, we may get an infinite number of facts where the variable is instantiated to different values.

So that these possibilities are avoided, Datalog rules are required to satisfy the following **safety** conditions:

1. Every variable that appears in the head of the rule also appears in a nonarithmetic positive literal in the body of the rule.
2. Every variable appearing in a negative literal in the body of the rule also appears in some positive literal in the body of the rule.

If all the rules in a nonrecursive Datalog program satisfy the preceding safety conditions, then all the view relations defined in the program can be shown to be finite, as long as all the database relations are finite. The conditions can be weakened somewhat to allow variables in the head to appear only in an arithmetic literal in the body in some cases. For example, in the rule

$$p(A) \text{ :- } q(B), A = B + 1$$

we can see that if relation q is finite, then so is p, according to the properties of addition, even though variable A appears in only an arithmetic literal.

27.4.5 Relational Operations in Datalog

Nonrecursive Datalog expressions without arithmetic operations are equivalent in expressive power to expressions using the basic operations in relational algebra ($\cup$, $-$, $\times$, σ, Π, and ρ). We shall not formally prove this assertion here. Rather, we shall show through examples how the various relational-algebra operations can be expressed in Datalog. In all cases, we define a view relation called *query* to illustrate the operations.

We have already seen how to do selection by using Datalog rules. We perform projections simply by using only the required attributes in the head of the rule. To project attribute *account_name* from account, we use

$$query(A) \text{ :- } account(A, N, B)$$

We can obtain the Cartesian product of two relations r_1 and r_2 in Datalog as follows:

$$query(X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m) \text{ :- } r_1(X_1, X_2, \ldots, X_n), r_2(Y_1, Y_2, \ldots, Y_m)$$

where r_1 is of arity n, and r_2 is of arity m, and the $X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m$ are all distinct variable names.

We form the union of two relations r_1 and r_2 (both of arity n) in this way:

$$query(X_1, X_2, \ldots, X_n) \text{ :- } r_1(X_1, X_2, \ldots, X_n)$$
$$query(X_1, X_2, \ldots, X_n) \text{ :- } r_2(X_1, X_2, \ldots, X_n)$$

We form the set difference of two relations r_1 and r_2 in this way:

$$query(X_1, X_2, \ldots, X_n) \text{ :- } r_1(X_1, X_2, \ldots, X_n), \textbf{ not } r_2(X_1, X_2, \ldots, X_n)$$

Finally, we note that with the positional notation used in Datalog, the renaming operator ρ is not needed. A relation can occur more than once in the rule body, but instead of renaming to give distinct names to the relation occurrences, we can use different variable names in the different occurrences.

It is possible to show that we can express any nonrecursive Datalog query without arithmetic by using the relational-algebra operations. We leave this demonstration as an exercise for you to carry out. You can thus establish the equivalence of the basic operations of relational algebra and nonrecursive Datalog without arithmetic operations.

Certain extensions to Datalog support the relational update operations (insertion, deletion, and update). The syntax for such operations varies from implementation to implementation. Some systems allow the use of + or − in rule heads to denote relational insertion and deletion. For example, we can move all accounts at the Perryridge branch to the Johnstown branch by executing

$$\begin{array}{l} +\, account(A, \text{“Johnstown”}, B) \;\text{:-}\; account(A, \text{“Perryridge”}, B) \\ -\, account(A, \text{“Perryridge”}, B) \;\text{:-}\; account(A, \text{“Perryridge”}, B) \end{array}$$

Some implementations of Datalog also support the aggregation operation of extended relational algebra. Again, there is no standard syntax for this operation.

27.4.6 Recursion in Datalog

Several database applications deal with structures that are similar to tree data structures. For example, consider employees in an organization. Some of the employees are managers. Each manager manages a set of people who report to him or her. But each of these people may in turn be managers, and they in turn may have other people who report to them. Thus, employees may be organized in a structure similar to a tree.

Suppose that we have a relation schema

$$Manager\_schema = (employee\_name, manager\_name)$$

Let *manager* be a relation on the preceding schema.

Suppose now that we want to find out which employees are supervised, directly or indirectly by a given manager—say, Jones. Thus, if the manager of Alon is Barinsky, and the manager of Barinsky is Estovar, and the manager of Estovar is Jones, then Alon, Barinsky, and Estovar are the employees controlled by Jones. People often write programs to manipulate tree data structures by recursion. Using the idea of recursion, we can define the set of employees controlled by Jones as follows: The people supervised by Jones are (1) people whose manager is Jones and (2) people whose manager is supervised by Jones. Note that case (2) is recursive.

We can encode the preceding recursive definition as a recursive Datalog view, called *empl_jones*:

$$\begin{array}{l} empl\_jones(X) \;\text{:-}\; manager(X, \text{“Jones”}\,) \\ empl\_jones(X) \;\text{:-}\; manager(X, Y), empl\_jones(Y) \end{array}$$

The first rule corresponds to case (1); the second rule corresponds to case (2). The view *empl_jones* depends on itself because of the second rule; hence, the preceding Datalog program is recursive. We *assume* that recursive Datalog programs contain no rules with

```
procedure Datalog-Fixpoint
    $I$ = set of facts in the database
    repeat
        $Old\_I = I$
        $I = I \cup infer(\mathcal{R}, I)$
    until $I = Old\_I$
```

Figure 27.10 Datalog-Fixpoint procedure.

negative literals. The reason will become clear later. The bibliographical notes refer to papers that describe where negation can be used in recursive Datalog programs.

The view relations of a recursive program that contains a set of rules $\mathcal{R}$ are defined to contain exactly the set of facts I computed by the iterative procedure Datalog-Fixpoint in Figure 27.10. The recursion in the Datalog program has been turned into an iteration in the procedure. At the end of the procedure, $infer(\mathcal{R}, I) \cup D = I$, where D is the set of facts in the database, and I is called a **fixed point** of the program.

Consider the program defining *empl_jones*, with the relation *manager*, as in Figure 27.11. The set of facts computed for the view relation *empl_jones* in each iteration appears in Figure 27.12. In each iteration, the program computes one more level of employees under Jones and adds it to the set *empl_jones*. The procedure terminates when there is no change to the set *empl_jones*, which the system detects by finding $I = Old\_I$. Such a termination point must be reached, since the set of managers and employees is finite. On the given *manager* relation, the procedure Datalog-Fixpoint terminates after iteration 4, when it detects that no new facts have been inferred.

You should verify that, at the end of the iteration, the view relation *empl_jones* contains exactly those employees who work under Jones. To print out the names of the employees supervised by Jones defined by the view, you can use the query

$$? \; empl\_jones(N)$$

employee_name	*manager_name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Figure 27.11 The *manager* relation.

Iteration_number	Tuples in *empl_jones*
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

Figure 27.12 Employees of Jones in iterations of procedure Datalog-Fixpoint.

To understand procedure Datalog-Fixpoint, we recall that a rule infers new facts from a given set of facts. Iteration starts with a set of facts I set to the facts in the database. These facts are all known to be true, but there may be other facts that are true as well.[1] Next, the set of rules $\mathcal{R}$ in the given Datalog program is used to infer what facts are true, given that facts in I are true. The inferred facts are added to I, and the rules are used again to make further inferences. This process is repeated until no new facts can be inferred.

For safe Datalog programs, we can show that there will be some point where no more new facts can be derived; that is, for some k, $I_{k+1} = I_k$. At this point, then, we have the final set of true facts. Further, given a Datalog program and a database, the fixed-point procedure infers all the facts that can be inferred to be true.

If a recursive program contains a rule with a negative literal, the following problem can arise. Recall that when we make an inference by using a ground instantiation of a rule, for each negative literal **not** q in the rule body we check that q is not present in the set of facts I. This test assumes that q cannot be inferred later. However, in the fixed-point iteration, the set of facts I grows in each iteration, and even if q is not present in I at one iteration, it may appear in I later. Thus, we may have made an inference in one iteration that can no longer be made at an earlier iteration, and the inference was incorrect. We require that a recursive program should not contain negative literals, in order to avoid such problems.

Instead of creating a view for the employees supervised by a specific manager Jones, we can create a more general view relation *empl* that contains every tuple (X, Y) such that X is directly or indirectly managed by Y, using the following program (also shown in Figure 27.7):

$$\begin{aligned} empl(X, Y) &\text{ :- } manager(X, Y) \\ empl(X, Y) &\text{ :- } manager(X, Z), empl(Z, Y) \end{aligned}$$

To find the direct and indirect subordinates of Jones, we simply use the query

[1]The word *fact* is used in a technical sense to note membership of a tuple in a relation. Thus, in the Datalog sense of "fact," a fact may be true (the tuple is indeed in the relation) or false (the tuple is not in the relation).

$$? \, empl(X, \text{“Jones”})$$

which gives the same set of values for X as the view *empl_jones*. Most Datalog implementations have sophisticated query optimizers and evaluation engines that can run the preceding query at about the same speed at which they could evaluate the view *empl _jones*.

The view *empl* defined previously is called the **transitive closure** of the relation *manager*. If the relation *manager* were replaced by any other binary relation R, the preceding program would define the transitive closure of R.

27.4.7 The Power of Recursion

Datalog with recursion has more expressive power than Datalog without recursion. In other words, there are queries on the database that we can answer by using recursion but cannot answer without using it. For example, we cannot express transitive closure in Datalog without using recursion (or for that matter, in SQL or QBE without recursion). Consider the transitive closure of the relation *manager*. Intuitively, a fixed number of joins can find only those employees that are some (other) fixed number of levels down from any manager (we will not attempt to prove this result here). Since any given nonrecursive query has a fixed number of joins, there is a limit on how many levels of employees the query can find. If the number of levels of employees in the *manager* relation is more than the limit of the query, the query will miss some levels of employees. Thus, a nonrecursive Datalog program cannot express transitive closure.

An alternative to recursion is to use an external mechanism, such as embedded SQL, to iterate on a nonrecursive query. The iteration in effect implements the fixed-point loop of Figure 27.10. In fact, that is how such queries are implemented on database systems that do not support recursion. However, writing such queries by iteration is more complicated than using recursion, and evaluation by recursion can be optimized to run faster than evaluation by iteration.

The expressive power provided by recursion must be used with care. It is relatively easy to write recursive programs that will generate an infinite number of facts, as this program illustrates:

$$\begin{array}{l} number(0) \\ number(A) \;\text{:-}\; number(B), A = B + 1 \end{array}$$

The program generates $number(n)$ for all positive integers n, which is infinite and will not terminate. The second rule of the program does not satisfy the safety condition in Section 27.4.4. Programs that satisfy the safety condition will terminate, even if they are recursive, provided that all database relations are finite. For such programs, tuples in view relations can contain only constants from the database, and hence the view relations must be finite. The converse is not true; that is, there are programs that do not satisfy the safety conditions but that do terminate.

The procedure Datalog-Fixpoint iteratively uses the function $\mathit{infer}(\mathcal{R}, I)$ to compute what facts are true, given a recursive Datalog program. Although we considered only the case of Datalog programs without negative literals, the procedure can also be used on views defined in other languages, such as SQL or relational algebra, provided that the views satisfy the conditions described next. Regardless of the language used to define a view V, the view can be thought of as being defined by an expression E_V that, given a set of facts I, returns a set of facts $E_V(I)$ for the view relation V. Given a set of view definitions $\mathcal{R}$ (in any language), we can define a function $\mathit{infer}(\mathcal{R}, I)$ that returns $I \cup \bigcup_{V \in \mathcal{R}} E_V(I)$. The preceding function has the same form as the *infer* function for Datalog.

A view V is said to be **monotonic** if, given any two sets of facts I_1 and I_2 such that $I_1 \subseteq I_2$, then $E_V(I_1) \subseteq E_V(I_2)$, where E_V is the expression used to define V. Similarly, the function *infer* is said to be monotonic if

$$I_1 \subseteq I_2 \Rightarrow \mathit{infer}(\mathcal{R}, I_1) \subseteq \mathit{infer}(\mathcal{R}, I_2)$$

Thus, if *infer* is monotonic, given a set of facts I_0 that is a subset of the true facts, we can be sure that all facts in $\mathit{infer}(\mathcal{R}, I_0)$ are also true. Using the same reasoning as in Section 27.4.6, we can then show that procedure Datalog-Fixpoint is sound (i.e., it computes only true facts), provided that the function *infer* is monotonic.

Relational-algebra expressions that use only the operators $\Pi, \sigma, \times, \bowtie, \cup, \cap$, or ρ are monotonic. Recursive views can be defined by using such expressions.

However, relational expressions that use the operator $-$ are not monotonic. For example, let $\mathit{manager}_1$ and $\mathit{manager}_2$ be relations with the same schema as the *manager* relation. Let

$$\begin{aligned} I_1 = \{\ & \mathit{manager}_1(\text{“Alon”, “Barinsky”}),\ \mathit{manager}_1(\text{“Barinsky”, “Estovar”}),\\ & \mathit{manager}_2(\text{“Alon”, “Barinsky”})\ \} \end{aligned}$$

and let

$$\begin{aligned} I_2 = \{\ & \mathit{manager}_1(\text{“Alon”, “Barinsky”}),\ \mathit{manager}_1(\text{“Barinsky”, “Estovar”}),\\ & \mathit{manager}_2(\text{“Alon”, “Barinsky”}),\ \mathit{manager}_2(\text{“Barinsky”, “Estovar”})\} \end{aligned}$$

Consider the expression $\mathit{manager}_1 - \mathit{manager}_2$. Now the result of the preceding expression on I_1 is (“Barinsky”, “Estovar”), whereas the result of the expression on I_2 is the empty relation. But $I_1 \subseteq I_2$; hence, the expression is not monotonic. Expressions using the grouping operation of extended relational algebra are also nonmonotonic.

The fixed-point technique does not work on recursive views defined with nonmonotonic expressions. However, there are instances where such views are useful, particularly for defining aggregates on “part-subpart” relationships. Such relationships define what subparts make up each part. Subparts themselves may have further subparts, and so on; hence, the relationships, like the manager relationship, have a natural recursive structure. An example of an aggregate query on such a structure would be to compute the total number of subparts of each part. Writing this query in Datalog or in SQL (without procedural extensions) would require the use of a recursive view on a nonmonotonic expression. The bibliographical notes provide references to research on defining such views.

It is possible to define some kinds of recursive queries without using views. For example, extended relational operations have been proposed to define transitive closure, and extensions to the SQL syntax to specify (generalized) transitive closure have been proposed. However, recursive view definitions provide more expressive power than do the other forms of recursive queries.

27.5 Summary

- 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.
- The tuple relational calculus and the domain relational calculus are terse, formal languages that are inappropriate for casual users of a database system. Commercial database systems, therefore, use languages with more “syntactic sugar.” We have considered two query languages: QBEand Datalog.
- Datalog is derived from Prolog, but unlike Prolog, it has a declarative semantics, making simple queries easier to write and query evaluation easier to optimize.
- Defining views is particularly easy in Datalog, and the recursive views that Datalog supports make it possible to write queries, such as transitive-closure queries, that cannot be written without recursion or iteration. However, no accepted standards exist for important features, such as grouping and aggregation, in Datalog. Datalog remains mainly a research language.

Review Terms

- Tuple relational calculus
- Domain relational calculus
- Safety of expressions
- Expressive power of languages
- Datalog
- Rules

- Uses
- Defines
- Positive literal
- Negative literal
- Fact
- Recursive view
- Nonrecursive view
- Instantiation
- Infer
- Semantics
- Safety
- Fixed point
- Transitive closure
- Monotonic view definition

Practice Exercises

27.1 Let the following relation schemas be given:

$$\begin{aligned} R &= (A, B, C) \\ S &= (D, E, F) \end{aligned}$$

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

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

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

employee (<u>*person_name*</u>, *street*, *city*)
works (<u>*person_name*</u>, *company_name*, *salary*)
company (<u>*company_name*</u>, *city*)
manages (<u>*person_name*</u>, *manager_name*)

Figure 27.13 Employee database.

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

27.4 Consider the relational database of Figure 27.13 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."

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

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

27.6 Consider the relational database of Figure 27.13 where the primary keys are underlined. Give an expression in Datalog for each of the following queries:

a. Find all employees who work (directly or indirectly) under the manager "Jones."

b. Find all cities of residence of all employees who work (directly or indirectly) under the manager "Jones."

c. Find all pairs of employees who have a (direct or indirect) manager in common.

d. Find all pairs of employees who have a (direct or indirect) manager in common and are at the same number of levels of supervision below the common manager.

27.7 Describe how an arbitrary Datalog rule can be expressed as an extended relational-algebra view.

Exercises

27.8 Consider the employee database of Figure 27.13. Give expressions in tuple relational calculus for each of the following queries:

a. Find the names of all employees who work for “FBC”.

b. Find the names and cities of residence of all employees who work for “FBC”.

c. Find the names, street addresses, and cities of residence of all employees who work for “FBC” 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 “FBC”.

g. Find all employees who earn more than every employee of “SBC”.

h. Assume that the companies may be located in several cities. Find all companies located in every city in which “SBC” is located.

27.9 Repeat Exercise 27.8, writing domain relational calculus queries instead of tuple relational calculus queries.

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

27.11 Repeat Exercise 27.10, writing SQL queries instead of relational-algebra expressions.

27.12 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. $r ⟖ s$

b. $r ⟗ s$

c. $r ⟕ s$

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

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

27.15 Repeat Exercise 27.8 using Datalog.

27.16 Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R. Give expressions in Datalog equivalent to each of the following queries:

a. $r_1 \cup r_2$

b. $r_1 \cap r_2$

c. $r_1 - r_2$

d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

27.17 Write an extended relational-algebra view equivalent to the Datalog rule

$$p(A, C, D) :\!-\; q1(A, B),\; q2(B, C),\; q3(4, B),\; D = B + 1.$$

Tools

The Coral system from the University of Wisconsin–Madison (research.cs.wisc.edu/coral) is an implementation of Datalog. The XSB system from Stony Brook University (xsb.sourceforge.net) is a widely used Prolog implementation that supports database querying; recall that Datalog is a nonprocedural subset of Prolog.

Further Reading

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 (1983)].

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.

Datalog programs that have both recursion and negation can be assigned a simple semantics if the negation is "stratified"—that is, if there is no recursion through negation. [Chandra and Harel (1982)] and [Apt and Pugin (1987)] discuss stratified negation. An important extension, called the *modular-stratification semantics*, which handles a class of recursive programs with negative literals, is discussed in [Ross (1990)]; an evaluation technique for such programs is described by [Ramakrishnan et al. (1992)].

Bibliography

[Apt and Pugin (1987)] K. R. Apt and J. M. Pugin, "Maintenance of Stratified Database Viewed as a Belief Revision System", In *Proc. of the ACM Symposium on Principles of Database Systems* (1987), pages 136–145.

[Chandra and Harel (1982)] A. K. Chandra and D. Harel, "Structure and Complexity of Relational Queries", *Journal of Computer and System Sciences*, Volume 15, Number 10 (1982), pages 99–128.

[Codd (1972)] E. F. Codd. "Further Normalization of the Data Base Relational Model", In *[Rustin (1972)]*, pages 33–64 (1972).

[Codd (1979)] E. F. Codd, "Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, Volume 4, Number 4 (1979), pages 397–434.

[Codd (1990)] E. F. Codd, *The Relational Model for Database Management: Version 2*, Addison Wesley (1990).

[Date (1983)] C. J. Date, "The Outer Join", In *Proc. of the International Conference on Databases*, John Wiley and Sons (1983), pages 76–106.

[Escobar-Molano et al. (1993)] M. Escobar-Molano, R. Hull, and D. Jacobs, "Safety and Translation of Calculus Queries with Scalar Functions", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), pages 253–264.

[Klug (1982)] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions", *Journal of the ACM*, Volume 29, Number 3 (1982), pages 699–717.

[Ramakrishnan et al. (1992)] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, *Controlling the Search in Bottom-up Evaluation* (1992).

[Ross (1990)] K. A. Ross, "Modular Stratification and Magic Sets for DATALOG Programs with Negation", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 161–171.

[Rustin (1972)] R. Rustin, *Data Base Systems*, Prentice Hall (1972).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.