



Other Relational Query Languages

In Chapter 6 we presented the relational algebra, which forms the basis of the widely used SQL query language. SQL was covered in great detail in Chapters 3 and 5. We also presented two more formal languages, the tuple relational calculus and the domain relational calculus, which are declarative query languages based on mathematical logic. These two formal languages form the basis for two more user-friendly languages, QBE and Datalog, that we study in this chapter.

Unlike SQL, QBE is a graphical language, where queries *look* like tables. QBE and its variants are widely used in database systems on personal computers. Datalog has a syntax modeled after the Prolog language. Although not used commercially at present, Datalog has been used in several research database systems.

For QBE and 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.

In this chapter we illustrate our concepts using a bank enterprise with the schema shown in Figure 2.15.

C.1 Query-by-Example

Query-by-Example (QBE) is the name of both a data-manipulation language and an early database system that included this language.

The QBE data-manipulation language has two distinctive features:

1. Unlike most query languages and programming languages, QBE has a **two-dimensional syntax**. Queries *look* like tables. A query in a one-dimensional language (for example, SQL) *can* be written in one (possibly long) line. A two-dimensional language *requires* two dimensions for its expression. (There is a one-dimensional version of QBE, but we shall not consider it in our discussion.)

2 Appendix C Other Relational Query Languages

2. QBE queries are expressed “by example.” Instead of giving a procedure for obtaining the desired answer, the user gives an example of what is desired. The system generalizes this example to compute the answer to the query.

Despite these unusual features, there is a close correspondence between QBE and the domain relational calculus.

There are two flavors of QBE: the original text-based version and a graphical version developed later that is supported by the Microsoft Access database system. In this section we provide a brief overview of the data-manipulation features of both versions of QBE. We first cover features of the text-based QBE that correspond to the SQL **select-from-where** clause without aggregation or updates. See the bibliographical notes for references where you can obtain more information about how the text-based version of QBE handles sorting of output, aggregation, and update.

C.1.1 Skeleton Tables

We express queries in QBE by **skeleton tables**. These tables show the relation schema, as in Figure C.1. Rather than clutter the display with all skeletons, the user selects those skeletons needed for a given query and fills in the skeletons with **example rows**. An example row consists of constants and *example elements*, which are domain variables. To avoid confusion between the two, QBE uses an underscore character (.) before domain variables, as in $_x$, and lets constants appear without any qualification. This convention is in contrast to those in most other languages, in which constants are quoted and variables appear without any qualification.

C.1.2 Queries on One Relation

Returning to our ongoing bank example, to find all loan numbers at the Perryridge branch, we bring up the skeleton for the *loan* relation, and fill it in as follows:

loan	loan_number	branch_name	amount
	P. $_x$	Perryridge	

This query tells the system to look for tuples in *loan* that have “Perryridge” as the value for the *branch_name* attribute. For each such tuple, the system assigns the value of the *loan_number* attribute to the variable x . It “prints” (actually, displays) the value of the variable x , because the command P. appears in the *loan_number* column next to the variable x . Observe that this result is similar to what would be done to answer the domain-relational-calculus query

$$\{\langle x \rangle \mid \exists b, a (\langle x, b, a \rangle \in \text{loan} \wedge b = \text{“Perryridge”})\}$$

QBE assumes that a blank position in a row contains a unique variable. As a result, if a variable does not appear more than once in a query, it may be omitted. Our previous query could thus be rewritten as

<i>branch</i>	<i>branch_name</i>	<i>branch_city</i>	<i>assets</i>

<i>customer</i>	<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>

<i>loan</i>	<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>

<i>borrower</i>	<i>customer_name</i>	<i>loan_number</i>

<i>account</i>	<i>account_number</i>	<i>branch_name</i>	<i>balance</i>

<i>depositor</i>	<i>customer_name</i>	<i>account_number</i>

Figure C.1 QBE skeleton tables for the bank example.

<i>loan</i>	<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
	P.	Perryridge	

QBE (unlike SQL) performs duplicate elimination automatically. To suppress duplicate elimination, we insert the command ALL. after the P. command:

<i>loan</i>	<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
	P.ALL.	Perryridge	

To display the entire *loan* relation, we can create a single row consisting of P. in every field. Alternatively, we can use a shorthand notation by placing a single P. in the column headed by the relation name:

4 Appendix C Other Relational Query Languages

<i>loan</i>	<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
P.			

QBE allows queries that involve arithmetic comparisons (for example, $>$), rather than equality comparisons, as in “Find the loan numbers of all loans with a loan amount of more than \$700”:

<i>loan</i>	<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
	P.		>700

Comparisons can involve only one arithmetic expression on the right-hand side of the comparison operation (for example, $> (_x + _y - 20)$). The expression can include both variables and constants. The space on the left-hand side of the comparison operation must be blank. The arithmetic operations that QBE supports are $=$, $<$, \leq , $>$, \geq , and \neg .

Note that requiring the left-hand side to be blank implies that we cannot compare two distinct named variables. We shall deal with this difficulty shortly.

As yet another example, consider the query “Find the names of all branches that are not located in Brooklyn.” This query can be written as follows:

<i>branch</i>	<i>branch_name</i>	<i>branch_city</i>	<i>assets</i>
	P.	\neg Brooklyn	

The primary purpose of variables in QBE is to force values of certain tuples to have the same value on certain attributes. Consider the query “Find the loan numbers of all loans made jointly to Smith and Jones”:

<i>borrower</i>	<i>customer_name</i>	<i>loan_number</i>
	Smith	P. $_x$
	Jones	$_x$

To execute this query, the system finds all pairs of tuples in *borrower* that agree on the *loan_number* attribute, where the value for the *customer_name* attribute is “Smith” for one tuple and “Jones” for the other. The system then displays the value of the *loan_number* attribute.

In the domain relational calculus, the query would be written as

$$\{\langle l \rangle \mid \exists x (\langle x, l \rangle \in \text{borrower} \wedge x = \text{“Smith”}) \\ \wedge \exists x (\langle x, l \rangle \in \text{borrower} \wedge x = \text{“Jones”})\}$$

As another example, consider the query “Find all customers who live in the same city as Jones”:

<i>customer</i>	<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>
	P. _x		_y
	Jones		_y

C.1.3 Queries on Several Relations

QBE allows queries that span several different relations (analogous to Cartesian product or natural join in the relational algebra). The connections among the various relations are achieved through variables that force certain tuples to have the same value on certain attributes. As an illustration, suppose that we want to find the names of all customers who have a loan from the Perryridge branch. This query can be written as

<i>loan</i>	<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
	_x	Perryridge	

<i>borrower</i>	<i>customer_name</i>	<i>loan_number</i>
	P. _y	_x

To evaluate the preceding query, the system finds tuples in *loan* with “Perryridge” as the value for the *branch_name* attribute. For each such tuple, the system finds tuples in *borrower* with the same value for the *loan_number* attribute as the *loan* tuple. It displays the values for the *customer_name* attribute.

We can use a technique similar to the preceding one to write the query “Find the names of all customers who have both an account and a loan at the bank”:

<i>depositor</i>	<i>customer_name</i>	<i>account_number</i>
	P. _x	

<i>borrower</i>	<i>customer_name</i>	<i>loan_number</i>
	_x	

Now consider the query “Find the names of all customers who have an account at the bank, but who do not have a loan from the bank.” We express queries that involve negation in QBE by placing a **not** sign (\neg) under the relation name and next to an example row:

<i>depositor</i>	<i>customer_name</i>	<i>account_number</i>
	P. _x	

<i>borrower</i>	<i>customer_name</i>	<i>loan_number</i>
\neg	_x	

6 Appendix C Other Relational Query Languages

Compare the preceding query with our earlier query “Find the names of all customers who have both an account and a loan at the bank.” The only difference is the \neg appearing next to the example row in the *borrower* skeleton. This difference, however, has a major effect on the processing of the query. QBE finds all x values for which

1. There is a tuple in the *depositor* relation whose *customer_name* is the domain variable x .
2. There is no tuple in the *borrower* relation whose *customer_name* is the same as in the domain variable x .

The \neg can be read as “there does not exist.”

The fact that we placed the \neg under the relation name, rather than under an attribute name, is important. A \neg under an attribute name is shorthand for \neq . Thus, to find all customers who have at least two accounts, we write

<i>depositor</i>	<i>customer_name</i>	<i>account_number</i>
	P. $_x$	$_y$
	$_x$	$\neg _y$

In English, the preceding query reads “Display all *customer_name* values that appear in at least two tuples, with the second tuple having an *account_number* different from the first.”

C.1.4 The Condition Box

At times, it is either inconvenient or impossible to express all the constraints on the domain variables within the skeleton tables. To overcome this difficulty, QBE includes a **condition box** feature that allows the expression of general constraints over any of the domain variables. QBE allows logical expressions to appear in a condition box. The logical operators are the words **and** and **or**, or the symbols “&” and “|”.

For example, the query “Find the loan numbers of all loans made to Smith, to Jones (or to both jointly)” can be written as

<i>borrower</i>	<i>customer_name</i>	<i>loan_number</i>
	$_n$	P. $_x$

<i>conditions</i>
$_n = \text{Smith or } _n = \text{Jones}$

It is possible to express the above query without using a condition box, by using P. in multiple rows. However, queries with P. in multiple rows are sometimes hard to understand, and are best avoided.

As yet another example, suppose that we modify the final query in Section C.1.3 to be “Find all customers who are not named ‘Jones’ and who have at least two accounts.” We want to include an “ $x \neq \text{Jones}$ ” constraint in this query. We do that by bringing up the condition box and entering the constraint “ $\neg x = \text{Jones}$ ”:

conditions
$\neg x = \text{Jones}$

Turning to another example, to find all account numbers with a balance between \$1300 and \$1500, we write

account	account_number	branch_name	balance
	P.		$\neg x$

conditions
$\neg x \geq 1300$
$\neg x \leq 1500$

As another example, consider the query “Find all branches that have assets greater than those of at least one branch located in Brooklyn.” This query can be written as

branch	branch_name	branch_city	assets
	P. $\neg x$	Brooklyn	$\neg y$ $\neg z$

conditions
$\neg y > \neg z$

QBE allows complex arithmetic expressions to appear in a condition box. We can write the query “Find all branches that have assets that are at least twice as large as the assets of one of the branches located in Brooklyn” much as we did in the preceding query, by modifying the condition box to

conditions
$\neg y \geq 2 * \neg z$

To find the account number of accounts with a balance between \$1300 and \$2000, but not exactly \$1500, we write

<i>account</i>	<i>account_number</i>	<i>branch_name</i>	<i>balance</i>
	P.		$_x$
<i>conditions</i>			
$_x = (\geq 1300 \text{ and } \leq 2000 \text{ and } \neg 1500)$			

QBE uses the **or** construct in an unconventional way to allow comparison with a set of constant values. To find all branches that are located in either Brooklyn or Queens, we write

<i>branch</i>	<i>branch_name</i>	<i>branch_city</i>	<i>assets</i>
	P.	$_x$	
<i>conditions</i>			
$_x = (\text{Brooklyn or Queens})$			

C.1.5 The Result Relation

The queries that we have written thus far have one characteristic in common: The results to be displayed appear in a single relation schema. If the result of a query includes attributes from several relation schemas, we need a mechanism to display the desired result in a single table. For this purpose, we can declare a temporary *result* relation that includes all the attributes of the result of the query. We print the desired result by including the command P. in only the *result* skeleton table.

As an illustration, consider the query “Find the *customer_name*, *account_number*, and *balance* for all accounts at the Perryridge branch.” In relational algebra, we would construct this query as follows:

1. Join *depositor* and *account*.
2. Project *customer_name*, *account_number*, and *balance*.

To construct the same query in QBE, we proceed as follows:

1. Create a skeleton table, called *result*, with attributes *customer_name*, *account_number*, and *balance*. The name of the newly created skeleton table (that is, *result*) must be different from any of the previously existing database relation names.
2. Write the query.

The resulting query is

account	account_number	branch_name	balance
	_y	Perryridge	_z

depositor	customer_name	account_number
	_x	_y

result	customer_name	account_number	balance
P.	_x	_y	_z

C.2 Microsoft Access

In this section, we survey the QBE version supported by Microsoft Access. While the original QBE was designed for a text-based display environment, Access QBE is designed for a graphical display environment, and accordingly is called **graphical query-by-example (QBE)**.

Figure C.2 shows a sample QBE query. The query can be described in English as “Find the *customer_name*, *account_number*, and *balance* for all accounts at the Perryridge branch.” Section C.1.5 showed how it is expressed in QBE.

A minor difference in the QBE version is that the attributes of a table are written one below the other, instead of horizontally. A more significant difference

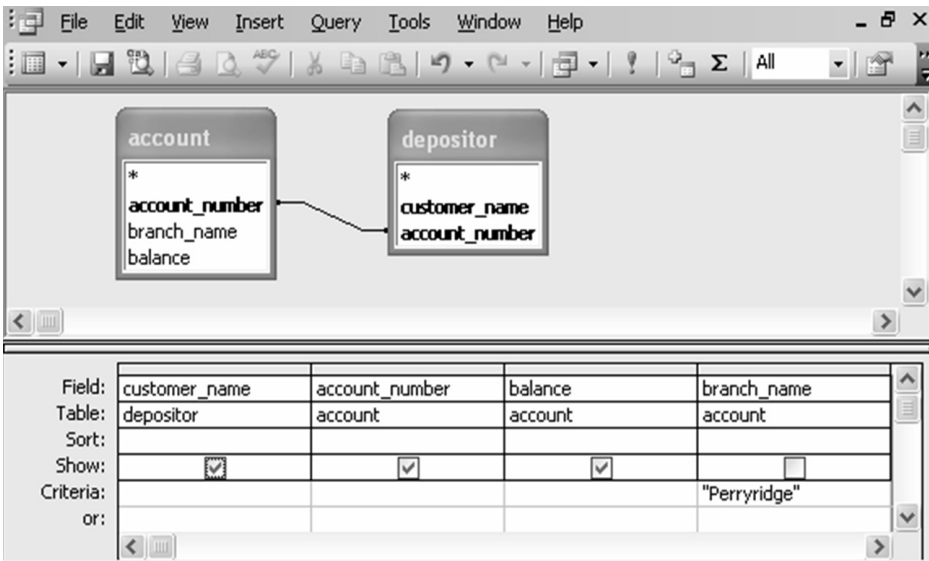


Figure C.2 An example query in Microsoft Access QBE.

is that the graphical version of QBE uses a line linking attributes of two tables, instead of a shared variable, to specify a join condition.

An interesting feature of QBE in Access is that links between tables are created automatically, on the basis of the attribute name. In the example in Figure C.2, the two tables *account* and *depositor* were added to the query. The attribute *account_number* is shared between the two selected tables, and the system automatically inserts a link between the two tables. In other words, a natural-join condition is imposed by default between the tables; the link can be deleted if it is not desired. The link can also be specified to denote a natural outer join, instead of a natural join.

Another minor difference in Access QBE is that it specifies attributes to be printed in a separate box, called the **design grid**, instead of using a P. in the table. It also specifies selections on attribute values in the design grid.

Queries involving group by and aggregation can be created in Access as shown in Figure C.3. The query in the figure finds the name, street, and city of all customers who have more than one account at the bank. The “group by” attributes as well as the aggregate functions are noted in the design grid.

Note that when a condition appears in a column of the design grid with the “Total” row set to an aggregate, the condition is applied on the aggregated value; for example, in Figure C.3, the selection “> 1” on the column *account_number* is applied on the result of the aggregate “Count.” Such selections correspond to selections in an SQL **having** clause.

Selection conditions can be applied on columns of the design grid that are neither grouped by nor aggregated; such attributes must be marked as “Where”

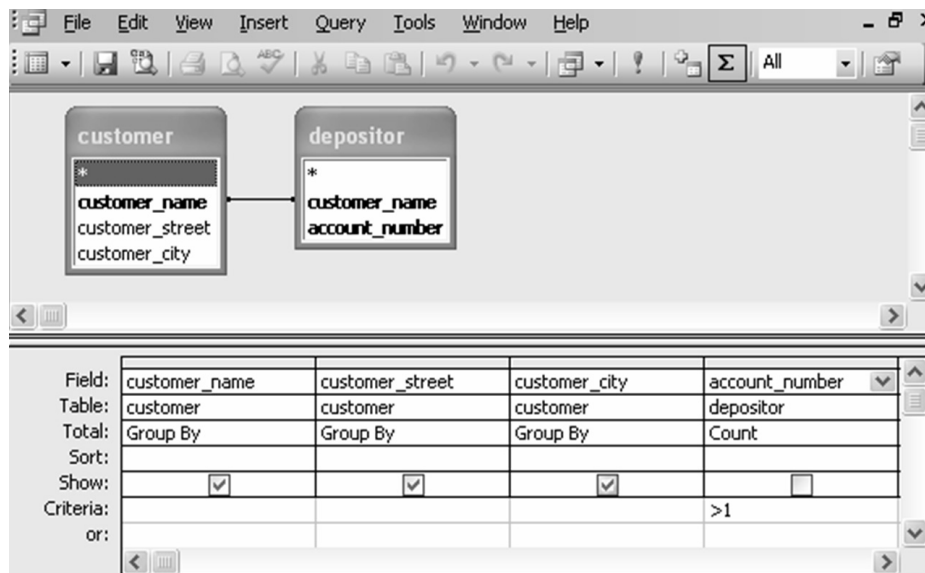


Figure C.3 An aggregation query in Microsoft Access QBE.

in the row “Total.” Such “Where” selections are applied before aggregation, and correspond to selections in an SQL **where** clause. However, such columns cannot be printed (marked as “Show”). Only columns where the “Total” row specifies either “group by,” or an aggregate function can be printed.

Queries are created through a graphical user interface, by first selecting tables. Attributes can then be added to the design grid by dragging and dropping them from the tables. Selection conditions, grouping, and aggregation can then be specified on the attributes in the design grid. Access QBE supports a number of other features too, including queries to modify the database through insertion, deletion, or update.

C.3 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.

C.3.1 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:

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

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:

```

for all A, B
if      (A, “Perryridge”, B) ∈ account and B > 700
then    (A, B) ∈ v1

```

Suppose that the relation *account* is as shown in Figure C.4. Then, the view relation *v1* contains the tuples in Figure C.5.

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

$$? v1(\text{“A-217”}, B)$$

<i>account_number</i>	<i>branch_name</i>	<i>balance</i>
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Perryridge	900
A-222	Redwood	700
A-217	Perryridge	750

Figure C.4 The *account* relation.

The answer to the query is

(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

(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:

interest_rate(A, 5) $:-$ *account*(A, N, B), B < 10000
interest_rate(A, 6) $:-$ *account*(A, N, B), B \geq 10000

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.

<i>account_number</i>	<i>balance</i>
A-201	900
A-217	750

Figure C.5 The *v1* relation.

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:

$$\begin{aligned} c(N) &:- \text{depositor}(N, A), \text{ not } \text{is_borrower}(N) \\ \text{is_borrower}(N) &:- \text{borrower}(N, L) \end{aligned}$$

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

$$\begin{aligned} v1(\text{account_number } A, \text{ balance } B) &:- \\ &\text{account}(\text{account_number } A, \text{ branch_name "Perryridge", balance } B), \\ &B > 700 \end{aligned}$$

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

C.3.2 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 C.3.3. 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

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

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

$$\text{not } p(t_1, t_2, \dots, t_n)$$

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

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

Literals involving arithmetic operations are treated specially. For example, the literal $B > 700$, although not in the syntax just described, can be conceptually

```

interest(A, I) :- account(A, "Perryridge", B),
                  interest_rate(A, R), I = B * R/100
interest_rate(A, 5) :- account(A, N, B), B < 10000
interest_rate(A, 6) :- account(A, N, B), B >= 10000

```

Figure C.6 Datalog program that defines interest on Perryridge accounts.

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 $>$. Clearly, 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 $z = x + y$.

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

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

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 C.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 above 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 .

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

Figure C.7 Recursive Datalog program.

In the example in Figure C.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**.

Consider the program in Figure C.7. Here, the view relation *empl* depends on itself (because of the second rule), and is therefore recursive. In contrast, the program in Figure C.6 is nonrecursive.

C.3.3 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 C.3.6. We define the semantics of a program by starting with the semantics of a single rule.

C.3.3.1 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:

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

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 ,

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

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 :

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

<i>account_number</i>	<i>balance</i>
A-201	900
A-217	750

Figure C.8 Result of $\text{infer}(R, I)$.

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

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

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

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

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

Suppose that we are given a set of facts I containing the tuples for relation *account* in Figure C.4. One possible instantiation of our running-example rule R is

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

The fact $\text{account}(\text{"A-217"}, \text{"Perryridge"}, 750)$ 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 $\text{infer}(R, I)$ has exactly the set of facts for $v1$ that appears in Figure C.8.

C.3.3.2 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

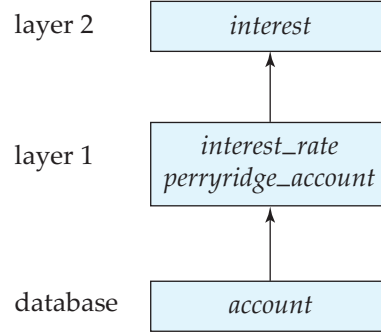


Figure C.9 Layering of view relations.

(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:

- 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 $1, 2, \dots, i$ and (2) all relations used in the bodies of rules defining p either are stored in the database or are in layers $1, 2, \dots, i$.

Consider the program in Figure C.6 with the additional rule:

$$\text{perryridge_account}(X, Y) \text{ :- } \text{account}(X, \text{"Perryridge"}, Y)$$

The layering of view relations in the program appears in Figure C.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 $1, 2, \dots, n$. 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 define I_1 as

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

- 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:

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

- 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 C.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 .

C.3.4 Safety

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

$$gt(X, Y) :- X > Y$$

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:

$$\text{not_in_loan}(L, B, A) :- \text{not loan}(L, B, A)$$

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

$$p(A) :- q(B), A = B + 1$$

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.

C.3.5 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

$$query(A) :- account(A, N, B)$$

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

$$query(X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_m) :- r_1(X_1, X_2, \dots, X_n), r_2(Y_1, Y_2, \dots, Y_m)$$

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

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

$$\begin{aligned} query(X_1, X_2, \dots, X_n) &:- r_1(X_1, X_2, \dots, X_n) \\ query(X_1, X_2, \dots, X_n) &:- r_2(X_1, X_2, \dots, X_n) \end{aligned}$$

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

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

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

$$\begin{aligned} + \text{account}(A, \text{"Johnstown"}, B) &:- \text{account}(A, \text{"Perryridge"}, B) \\ - \text{account}(A, \text{"Perryridge"}, B) &:- \text{account}(A, \text{"Perryridge"}, B) \end{aligned}$$

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

C.3.6 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

$$\text{Manager_schema} = (\text{employee_name}, \text{manager_name})$$

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*:

$$\begin{aligned} \text{empl_jones}(X) &:- \text{manager}(X, \text{"Jones"}) \\ \text{empl_jones}(X) &:- \text{manager}(X, Y), \text{empl_jones}(Y) \end{aligned}$$

```

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 C.10 Datalog-Fixpoint procedure.

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 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 C.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 C.11. The set of facts computed for the view relation *empl_jones* in each iteration appears in Figure C.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

<i>employee_name</i>	<i>manager_name</i>
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Figure C.11 The *manager* relation.

Iteration number	Tuples in <i>empl_jones</i>
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

Figure C.12 Employees of Jones in iterations of procedure Datalog-Fixpoint.

of the employees supervised by Jones defined by the view, you can use the query

$? \text{empl_jones}(N)$

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

¹The 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).

(X, Y) such that X is directly or indirectly managed by Y , using the following program (also shown in Figure C.7):

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

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

$$? \text{empl}(X, \text{"Jones"})$$

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 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 .

C.3.7 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 C.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:

$$\begin{aligned} \text{number}(0) \\ \text{number}(A) &:- \text{number}(B), A = B + 1 \end{aligned}$$

The program generates $number(n)$ for all positive integers n , which is clearly infinite, and will not terminate. The second rule of the program does not satisfy the safety condition in Section C.3.4. 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 $infer(\mathcal{R}, I)$ 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 $infer(\mathcal{R}, I)$ 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

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

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 $infer(\mathcal{R}, I_0)$ are also true. Using the same reasoning as in Section C.3.6, we can then show that procedure Datalog-Fixpoint is sound (that is, it computes only true facts), provided that the function $infer$ is monotonic.

Relational-algebra expressions that use only the operators Π , σ , \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 $manager_1$ and $manager_2$ be relations with the same schema as the $manager$ relation. Let

$$I_1 = \{ manager_1(\text{"Alon"}, \text{"Barinsky"}), manager_1(\text{"Barinsky"}, \text{"Estovar"}), \\ manager_2(\text{"Alon"}, \text{"Barinsky"}) \}$$

and let

$$I_2 = \{ manager_1(\text{"Alon"}, \text{"Barinsky"}), manager_1(\text{"Barinsky"}, \text{"Estovar"}), \\ manager_2(\text{"Alon"}, \text{"Barinsky"}), manager_2(\text{"Barinsky"}, \text{"Estovar"}) \}$$

Consider the expression $manager_1 - manager_2$. 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.

C.4 Summary

- 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: QBE and Datalog.
- QBE is based on a visual paradigm: The queries look much like tables.
- QBE and its variants have become popular with nonexpert database users because of the intuitive simplicity of the visual paradigm. The widely used Microsoft Access database system supports a graphical version of QBE, called GQBE.
- 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
- Query-by-Example (QBE)
- Two-dimensional syntax
- Skeleton tables
- Example rows
- Condition box
- Result relation
- Microsoft Access
- Graphical Query-By-Example (GQBE)
- Design grid
- Datalog
- Rules
- Uses
- Defines
- Positive literal
- Negative literal
- Fact
- Rule
 - Head
 - Body
- Datalog program
- Depend on
 - Directly
 - Indirectly
- Recursive view
- Nonrecursive view
- Instantiation
 - Ground instantiation
 - Satisfied
- Infer
- Semantics
 - Of a rule
 - Of a program
- Safety
- Fixed point
- Transitive closure
- Monotonic view definition

Practice Exercises

C.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. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$
- C.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:
- $\Pi_A(r_1)$
 - $\sigma_{B=17}(r_1)$
 - $r_1 \cup r_2$
 - $r_1 \cap r_2$
 - $r_1 - r_2$
 - $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$
- C.3** Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write expressions in QBE and Datalog for each of the following queries:
- $\{ \langle a \rangle \mid \exists b (\langle a, b \rangle \in r \wedge b = 7) \}$
 - $\{ \langle a, b, c \rangle \mid \langle a, b \rangle \in r \wedge \langle a, c \rangle \in s \}$
 - $\{ \langle a \rangle \mid \exists c (\langle a, c \rangle \in s \wedge \exists b_1, b_2 (\langle a, b_1 \rangle \in r \wedge \langle c, b_2 \rangle \in r \wedge b_1 > b_2)) \}$
- C.4** Consider the relational database of Figure C.13 where the primary keys are underlined. Give an expression in Datalog for each of the following queries:
- Find all employees who work (directly or indirectly) under the manager “Jones.”
 - Find all cities of residence of all employees who work (directly or indirectly) under the manager “Jones.”
 - Find all pairs of employees who have a (direct or indirect) manager in common.
 - 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.
- C.5** Describe how an arbitrary Datalog rule can be expressed as an extended relational-algebra view.

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

Figure C.13 Employee database.

Exercises

- C.6** Consider the employee database of Figure C.13. Give expressions in tuple relational calculus and domain relational calculus for each of the following queries:
- Find the names of all employees who work for First Bank Corporation.
 - Find the names and cities of residence of all employees who work for First Bank Corporation.
 - Find the names, street addresses, and cities of residence of all employees who work for First Bank Corporation and earn more than \$10,000 per annum.
 - Find all employees who live in the same city as that in which the company for which they work is located.
 - Find all employees who live in the same city and on the same street as their managers.
 - Find all employees in the database who do not work for First Bank Corporation.
 - Find all employees who earn more than every employee of Small Bank Corporation.
 - Assume that the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.
- C.7** 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:
- $\{ \langle a \rangle \mid \exists b (\langle a, b \rangle \in r \wedge b = 17) \}$
 - $\{ \langle a, b, c \rangle \mid \langle a, b \rangle \in r \wedge \langle a, c \rangle \in s \}$
 - $\{ \langle a \rangle \mid \exists b (\langle a, b \rangle \in r) \vee \forall c (\exists d (\langle d, c \rangle \in s) \Rightarrow \langle a, c \rangle \in s) \}$
 - $\{ \langle a \rangle \mid \exists c (\langle a, c \rangle \in s \wedge \exists b_1, b_2 (\langle a, b_1 \rangle \in r \wedge \langle c, b_2 \rangle \in r \wedge b_1 > b_2)) \}$

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*driver_id*, *license*, *report_number*, *damage_amount*)

Figure C.14 Insurance database.

- C.8** Repeat Exercise C.7, writing SQL queries instead of relational-algebra expressions.
- C.9** 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:
- $r \bowtie_{\sqsubset} s$
 - $r \bowtie_{\supset} s$
 - $r \bowtie s$
- C.10** Consider the insurance database of Figure C.14, where the primary keys are underlined. Construct the following GQBE queries for this relational database.
- Find the total number of people who owned cars that were involved in accidents in 1989.
 - Find the number of accidents in which the cars belonging to “John Smith” were involved.
- C.11** Give a tuple-relational-calculus expression to find the maximum value in relation $r(A)$.
- C.12** Repeat Exercise C.6 using QBE and Datalog.
- C.13** Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R . Give expressions in QBE and Datalog equivalent to each of the following queries:
- $r_1 \cup r_2$
 - $r_1 \cap r_2$
 - $r_1 - r_2$
 - $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$
- C.14** Write an extended relational-algebra view equivalent to the Datalog rule

$$p(A, C, D) :- q^1(A, B), q^2(B, C), q^3(4, B), D = B + 1$$

Tools

The Microsoft Access QBE is currently the most widely available implementation of QBE. The QMF and Everywhere editions of IBM DB2 also support QBE.

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

Bibliographical Notes

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.

The QBE database system was developed at IBM’s T. J. Watson Research Center in the early 1970s. The QBE data-manipulation language was later used in IBM’s Query Management Facility (QMF). The original version of Query-by-Example is described in Zloof [1977]. Other QBE implementations include Microsoft Access, and Borland Paradox (which is no longer supported).

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].