APPENDIX C

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. 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

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

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

branch	*branch_name*	*branch_city*	*assets*

customer	*customer_name*	*customer_street*	*customer_city*

loan	*loan_number*	*branch_name*	*amount*

borrower	*customer_name*	*loan_number*

account	*account_number*	*branch_name*	*balance*

depositor	*customer_name*	*account_number*

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

loan	*loan_number*	*branch_name*	*amount*
	P.	Perryridge	

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

loan	*loan_number*	*branch_name*	*amount*
	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:

loan	*loan_number*	*branch_name*	*amount*
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":

loan	*loan_number*	*branch_name*	*amount*
	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:

branch	*branch_name*	*branch_city*	*assets*
	P.	¬ 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":

borrower	*customer_name*	*loan_number*
	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

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

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

customer	*customer_name*	*customer_street*	*customer_city*
	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

loan	*loan_number*	*branch_name*	*amount*
	_*x*	Perryridge	

borrower	*customer_name*	*loan_number*
	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":

depositor	*customer_name*	*account_number*
	P._*x*	

borrower	*customer_name*	*loan_number*
	_*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 (¬) under the relation name and next to an example row:

depositor	*customer_name*	*account_number*
	P._*x*	

borrower	*customer_name*	*loan_number*
¬	_*x*	

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 ¬ 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 ¬ can be read as "there does not exist."

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

depositor	*customer_name*	*account_number*
	P._*x*	_*y*
	_*x*	¬ _*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

borrower	*customer_name*	*loan_number*
	_*n*	P._*x*

conditions
_*n* = Smith **or** _*n* = 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 "$x \neq$ Jones" constraint in this query. We do that by bringing up the condition box and entering the constraint "$\_x \neg =$ Jones":

conditions
$\_x \neg =$ 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.		$\_x$

conditions
$\_x \geq 1300$ $\_x \leq 1500$

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.$\_x$		$\_y$
		Brooklyn	$\_z$

conditions
$\_y > \_z$

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
$\_y \geq 2 * \_z$

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

account	*account_number*	*branch_name*	*balance*
	P.		_*x*

conditions
_*x* = (≥ 1300 **and** ≤ 2000 **and** ¬ 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

branch	*branch_name*	*branch_city*	*assets*
	P.	_*x*	

conditions
_*x* = (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*

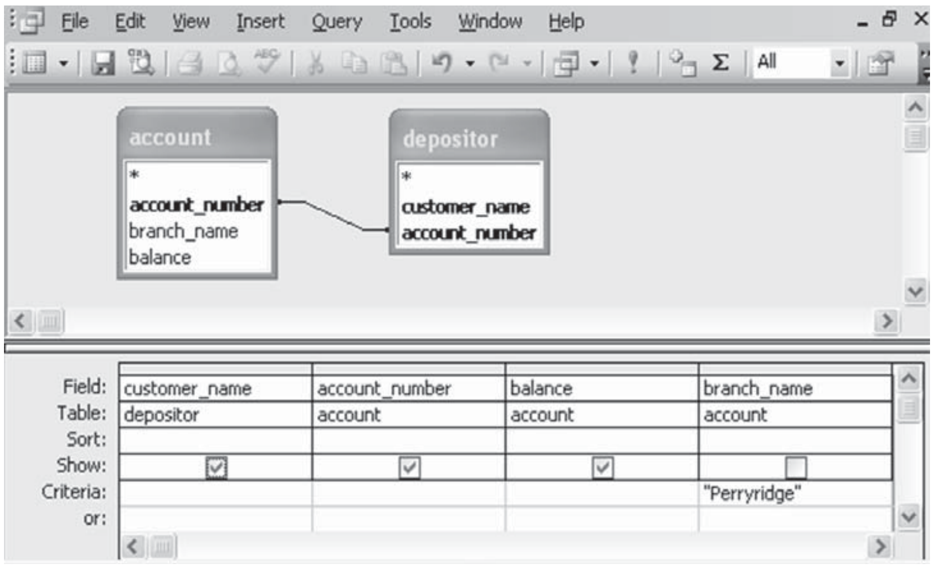
result	*customer_name*	*account_number*	*balance*
P.	_*x*	_*y*	_*z*

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 (GQBE)**.

Figure C.2 shows a sample GQBE 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 GQBE version is that the attributes of a table are written one below the other, instead of horizontally. A more significant difference



Field:	customer_name	account_number	balance	branch_name
Table:	depositor	account	account	account
Sort:				
Show:	✓	✓	✓	☐
Criteria:				"Perryridge"
or:				

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.

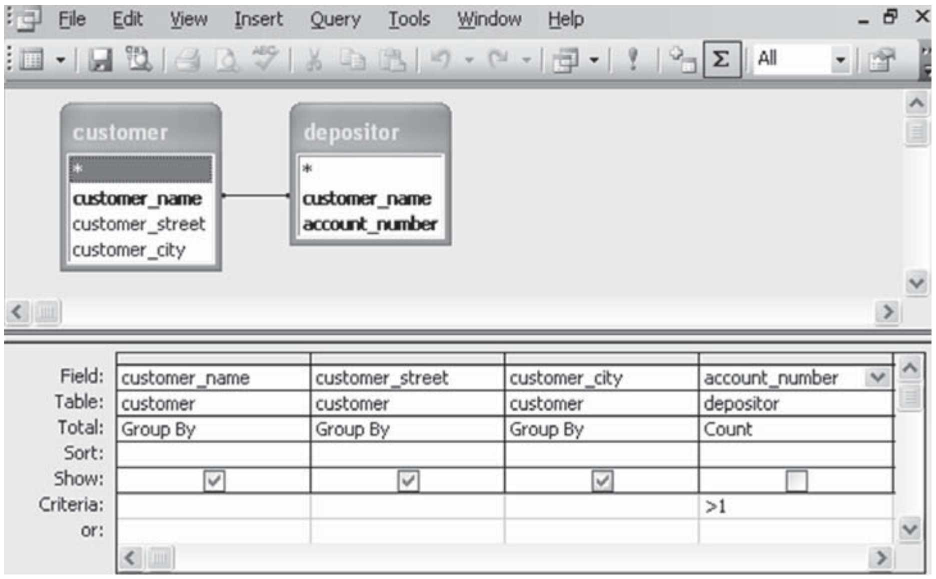
An interesting feature of QBE in Access is that links between tables are created automatically, on the basis of the attribute name. In the example in Figure C.2, the two tables *account* and *depositor* were added to the query. The attribute *account_number* is shared between the two selected tables, 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"



Field:	customer_name	customer_street	customer_city	account_number
Table:	customer	customer	customer	depositor
Total:	Group By	Group By	Group By	Count
Sort:				
Show:	☑	☑	☑	☐
Criteria:				>1
or:				

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:

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

Suppose that the relation *account* is as shown in Figure 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:

$$?\ v1(\text{“A-217”}, B)$$