6

QUERY-BY-EXAMPLE (QBE)

Example is always more efficacious than precept.

—Samuel Johnson

6.1 INTRODUCTION

Query-by-Example (QBE) is another language for querying (and, like SQL, for creating and modifying) relational data. It is different from SQL, and from most other database query languages, in having a graphical user interface that allows users to write queries by creating *example tables* on the screen. A user needs minimal information to get started and the whole language contains relatively few concepts. QBE is especially suited for queries that are not too complex and can be expressed in terms of a few tables.

QBE, like SQL, was developed at IBM and QBE is an IBM trademark, but a number of other companies sell QBE-like interfaces, including Paradox. Some systems, such as Microsoft Access, offer partial support for form-based queries and reflect the influence of QBE. Often a QBE-like interface is offered in addition to SQL, with QBE serving as a more intuitive user-interface for simpler queries and the full power of SQL available for more complex queries. An appreciation of the features of QBE offers insight into the more general, and widely used, paradigm of tabular query interfaces for relational databases.

This presentation is based on IBM's Query Management Facility (QMF) and the QBE version that it supports (Version 2, Release 4). This chapter explains how a tabular interface can provide the expressive power of relational calculus (and more) in a user-friendly form. The reader should concentrate on the connection between QBE and domain relational calculus (DRC), and the role of various important constructs (e.g., the conditions box), rather than on QBE-specific details. We note that every QBE query can be expressed in SQL; in fact, QMF supports a command called `CONVERT` that generates an SQL query from a QBE query.

We will present a number of example queries using the following schema:

Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)

Boats(*bid:* `integer`, *bname:* `string`, *color:* `string`)
Reserves(*sid:* `integer`, *bid:* `integer`, *day:* `dates`)

The key fields are underlined, and the domain of each field is listed after the field name.

We introduce QBE queries in Section 6.2 and consider queries over multiple relations in Section 6.3. We consider queries with set-difference in Section 6.4 and queries with aggregation in Section 6.5. We discuss how to specify complex constraints in Section 6.6. We show how additional computed fields can be included in the answer in Section 6.7. We discuss update operations in QBE in Section 6.8. Finally, we consider relational completeness of QBE and illustrate some of the subtleties of QBE queries with negation in Section 6.9.

6.2 BASIC QBE QUERIES

A user writes queries by creating *example tables*. QBE uses *domain variables*, as in the DRC, to create example tables. The domain of a variable is determined by the column in which it appears, and variable symbols are prefixed with underscore (_) to distinguish them from constants. Constants, including strings, appear unquoted, in contrast to SQL. The fields that should appear in the answer are specified by using the command `P.`, which stands for *print*. The fields containing this command are analogous to the *target-list* in the `SELECT` clause of an SQL query.

We introduce QBE through example queries involving just one relation. To print the names and ages of all sailors, we would create the following example table:

Sailors	*sid*	*sname*	*rating*	*age*
		P._N		P._A

A variable that appears only once can be omitted; QBE supplies a unique new name internally. Thus the previous query could also be written by omitting the variables _N and _A, leaving just `P.` in the *sname* and *age* columns. The query corresponds to the following DRC query, obtained from the QBE query by introducing existentially quantified domain variables for each field.

$$\{\langle N, A\rangle \mid \exists I, T(\langle I, N, T, A\rangle \in Sailors)\}$$

A large class of QBE queries can be translated to DRC in a direct manner. (Of course, queries containing features such as aggregate operators cannot be expressed in DRC.) We will present DRC versions of several QBE queries. Although we will not define the translation from QBE to DRC formally, the idea should be clear from the examples;

intuitively, there is a term in the DRC query for each row in the QBE query, and the terms are connected using $\wedge$.[1]

A convenient shorthand notation is that if we want to print all fields in some relation, we can place `P.` under the name of the relation. This notation is like the `SELECT *` convention in SQL. It is equivalent to placing a `P.` in every field:

Sailors	*sid*	*sname*	*rating*	*age*
P.				

Selections are expressed by placing a constant in some field:

Sailors	*sid*	*sname*	*rating*	*age*
P.			10	

Placing a constant, say 10, in a column is the same as placing the condition *=10*. This query is very similar in form to the equivalent DRC query

$$\{\langle I, N, 10, A\rangle \mid \langle I, N, 10, A\rangle \in Sailors\}$$

We can use other comparison operations ($<, >, <=, >=, \neg$) as well. For example, we could say < 10 to retrieve sailors with a rating less than 10 or say $\neg 10$ to retrieve sailors whose rating is not equal to 10. The expression $\neg 10$ in an attribute column is the same as $\neq 10$. As we will see shortly, $\neg$ under the relation name denotes (a limited form of) $\neg\exists$ in the relational calculus sense.

6.2.1 Other Features: Duplicates, Ordering Answers

We can explicitly specify whether duplicate tuples in the answer are to be eliminated (or not) by putting `UNQ.` (respectively `ALL.`) under the relation name.

We can order the presentation of the answers through the use of the `.AO` (for *ascending order*) and `.DO` commands in conjunction with `P.` An optional integer argument allows us to sort on more than one field. For example, we can display the names, ages, and ratings of all sailors in ascending order by age, and for each age, in ascending order by rating as follows:

Sailors	*sid*	*sname*	*rating*	*age*
		P.	P.AO(2)	P.AO(1)

[1] The semantics of QBE is unclear when there are several rows containing `P.` or if there are rows that are not linked via shared variables to the row containing `P.` We will discuss such queries in Section 6.6.1.

6.3 QUERIES OVER MULTIPLE RELATIONS

To find sailors with a reservation, we have to combine information from the Sailors and the Reserves relations. In particular we have to select tuples from the two relations with the same value in the join column *sid*. We do this by placing the same variable in the *sid* columns of the two example relations.

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._S		

Reserves	*sid*	*bid*	*day*
	_Id		

To find sailors who have reserved a boat for 8/24/96 and who are older than 25, we could write:[2]

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._S		> 25

Reserves	*sid*	*bid*	*day*
	_Id		'8/24/96'

Extending this example, we could try to find the colors of Interlake boats reserved by sailors who have reserved a boat for 8/24/96 and who are older than 25:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id			> 25

Reserves	*sid*	*bid*	*day*
	_Id	_B	'8/24/96'

Boats	*bid*	*bname*	*color*
	_B	Interlake	P.

As another example, the following query prints the names and ages of sailors who have reserved some boat that is also reserved by the sailor with id 22:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._N		

Reserves	*sid*	*bid*	*day*
	_Id	_B	
	22	_B	

Each of the queries in this section can be expressed in DRC. For example, the previous query can be written as follows:

$$\{\langle N\rangle \mid \exists Id, T, A, B, D1, D2(\langle Id, N, T, A\rangle \in Sailors$$
$$\wedge\langle Id, B, D1\rangle \in Reserves \wedge \langle 22, B, D2\rangle \in Reserves)\}$$

[2]Incidentally, note that we have quoted the date value. In general, constants are not quoted in QBE. The exceptions to this rule include date values and string values with embedded blanks or special characters.

Notice how the only free variable (N) is handled and how Id and B are repeated, as in the QBE query.

6.4 NEGATION IN THE RELATION-NAME COLUMN

We can print the names of sailors who do *not* have a reservation by using the ¬ command in the relation name column:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._S		

Reserves	*sid*	*bid*	*day*
¬	_Id		

This query can be read as follows: "Print the *sname* field of Sailors tuples such that there is *no* tuple in Reserves with the same value in the *sid* field." Note the importance of *sid* being a key for Sailors. In the relational model, keys are the only available means for *unique identification* (of sailors, in this case). (Consider how the meaning of this query would change if the Reserves schema contained *sname*—which is not a key!—rather than *sid*, and we used a common variable in this column to effect the join.)

All variables in a negative row (i.e., a row that is preceded by ¬) must also appear in positive rows (i.e., rows not preceded by ¬). Intuitively, variables in positive rows can be instantiated in many ways, based on the tuples in the input instances of the relations, and each negative row involves a simple check to see if the corresponding relation contains a tuple with certain given field values.

The use of ¬ in the relation-name column gives us a limited form of the set-difference operator of relational algebra. For example, we can easily modify the previous query to find sailors who are not (both) younger than 30 and rated higher than 4:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._S		

Sailors	*sid*	*sname*	*rating*	*age*
¬	_Id		> 4	< 30

This mechanism is not as general as set-difference, because there is no way to control the order in which occurrences of ¬ are considered if a query contains more than one occurrence of ¬. To capture full set-difference, views can be used. (The issue of QBE's relational completeness, and in particular the ordering problem, is discussed further in Section 6.9.)

6.5 AGGREGATES

Like SQL, QBE supports the aggregate operations `AVG.`, `COUNT.`, `MAX.`, `MIN.`, and `SUM.` By default, these aggregate operators do *not* eliminate duplicates, with the exception

of COUNT., which does eliminate duplicates. To eliminate duplicate values, the variants AVG.UNQ. and SUM.UNQ. must be used. (Of course, this is irrelevant for MIN. and MAX.) Curiously, there is no variant of COUNT. that does *not* eliminate duplicates.

Consider the instance of Sailors shown in Figure 6.1. On this instance the following

sid	*sname*	*rating*	*age*
22	dustin	7	45.0
58	rusty	10	35.0
44	horatio	7	35.0

Figure 6.1 An Instance of Sailors

query prints the value 38.3:

Sailors	*sid*	*sname*	*rating*	*age*	
				_A	P.AVG._A

Thus, the value 35.0 is counted twice in computing the average. To count each age only once, we could specify P.AVG.UNQ. instead, and we would get 40.0.

QBE supports *grouping*, as in SQL, through the use of the G. command. To print average ages by rating, we could use:

Sailors	*sid*	*sname*	*rating*	*age*	
			G.P.	_A	P.AVG._A

To print the answers in sorted order by rating, we could use G.P.AO or G.P.DO. instead. When an aggregate operation is used in conjunction with P., or there is a use of the G. operator, every column to be printed must specify either an aggregate operation or the G. operator. (Note that SQL has a similar restriction.) If G. appears in more than one column, the result is similar to placing each of these column names in the GROUP BY clause of an SQL query. If we place G. in the *sname* and *rating* columns, all tuples in each group have the same *sname* value and also the same *rating* value.

We consider some more examples using aggregate operations after introducing the conditions box feature.

6.6 THE CONDITIONS BOX

Simple conditions can be expressed directly in columns of the example tables. For more complex conditions QBE provides a feature called a **conditions box**.

Conditions boxes are used to do the following:

- *Express a condition involving two or more columns*, such as *_R/_A > 0.2*.
- *Express a condition involving an aggregate operation on a group*, for example, `AVG._A > 30`. Notice that this use of a conditions box is similar to the `HAVING` clause in SQL. The following query prints those ratings for which the average age is more than 30:

Sailors	*sid*	*sname*	*rating*	*age*	*Conditions*
			`G.P.`	_A	`AVG._A` > 30

As another example, the following query prints the *sid*s of sailors who have reserved all boats for which there is some reservation:

Sailors	*sid*	*sname*	*rating*	*age*
	`P.G._Id`			

Reserves	*sid*	*bid*	*day*	*Conditions*
	_Id	_B1 _B2		`COUNT._B1` = `COUNT._B2`

For each _Id value (notice the `G.` operator), we count all _B1 values to get the number of (distinct) *bid* values reserved by sailor _Id. We compare this count against the count of all _B2 values, which is simply the total number of (distinct) *bid* values in the Reserves relation (i.e., the number of boats with reservations). If these counts are equal, the sailor has reserved all boats for which there is some reservation. Incidentally, the following query, intended to print the names of such sailors, is incorrect:

Sailors	*sid*	*sname*	*rating*	*age*
	`P.G._Id`	`P.`		

Reserves	*sid*	*bid*	*day*	*Conditions*
	_Id	_B1 _B2		`COUNT._B1` = `COUNT._B2`

The problem is that in conjunction with G., only columns with either G. or an aggregate operation can be printed. This limitation is a direct consequence of the SQL definition of GROUPBY, which we discussed in Section 5.5.1; QBE is typically implemented by translating queries into SQL. If P.G. replaces P. in the *sname* column, the query is legal, and we then group by both *sid* and *sname*, which results in the same groups as before because *sid* is a key for Sailors.

- *Express conditions involving the* AND *and* OR *operators.* We can print the names of sailors who are younger than 20 *or* older than 30 as follows:

Sailors	*sid*	*sname*	*rating*	*age*	*Conditions*
		P.		_A	_A < 20 OR 30 < _A

We can print the names of sailors who are both younger than 20 *and* older than 30 by simply replacing the condition with $\_A < 20$ AND $30 < \_A$; of course, the set of such sailors is always empty! We can print the names of sailors who are either older than 20 *or* have a rating equal to 8 by using the condition $20 < \_A$ OR $\_R = 8$, and placing the variable _R in the *rating* column of the example table.

6.6.1 And/Or Queries

It is instructive to consider how queries involving AND and OR can be expressed in QBE without using a conditions box. We can print the names of sailors who are younger than 30 *or* older than 20 by simply creating two example rows:

Sailors	*sid*	*sname*	*rating*	*age*
		P.		< 30
		P.		> 20

To translate a QBE query with several rows containing P., we create subformulas for each row with a P. and connect the subformulas through $\vee$. If a row containing P. is linked to other rows through shared variables (which is not the case in this example), the subformula contains a term for each linked row, all connected using $\wedge$. Notice how the answer variable N, which must be a free variable, is handled:

$$\{\langle N\rangle \mid \exists I1, N1, T1, A1, I2, N2, T2, A2($$
$$\langle I1, N1, T1, A1\rangle \in Sailors(A1 < 30 \wedge N = N1)$$
$$\vee\langle I2, N2, T2, A2\rangle \in Sailors(A2 > 20 \wedge N = N2))\}$$

To print the names of sailors who are both younger than 30 *and* older than 20, we use the same variable in the key fields of both rows:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P.		< 30
	_Id			> 20

The DRC formula for this query contains a term for each linked row, and these terms are connected using $\wedge$:

$$\{\langle N\rangle \mid \exists I1, N1, T1, A1, N2, T2, A2$$
$$(\langle I1, N1, T1, A1\rangle \in Sailors(A1 < 30 \wedge N = N1)$$
$$\wedge\langle I1, N2, T2, A2\rangle \in Sailors(A2 > 20 \wedge N = N2))\}$$

Compare this DRC query with the DRC version of the previous query to see how closely they are related (and how closely QBE follows DRC).

6.7 UNNAMED COLUMNS

If we want to display some information in addition to fields retrieved from a relation, we can create *unnamed columns* for display.[3] As an example—admittedly, a silly one!—we could print the name of each sailor along with the ratio *rating/age* as follows:

Sailors	*sid*	*sname*	*rating*	*age*	
		P.	_R	_A	P._R / _A

All our examples thus far have included P. commands in exactly one table. This is a QBE restriction. If we want to display fields from more than one table, we have to use unnamed columns. To print the names of sailors along with the dates on which they have a boat reserved, we could use the following:

Sailors	*sid*	*sname*	*rating*	*age*		*Reserves*	*sid*	*bid*	*day*
	_Id	P.			P._D		_Id		_D

Note that unnamed columns should not be used for expressing *conditions* such as *_D >8/9/96*; a conditions box should be used instead.

6.8 UPDATES

Insertion, deletion, and modification of a tuple are specified through the commands I., D., and U., respectively. We can insert a new tuple into the Sailors relation as follows:

[3] A QBE facility includes simple commands for drawing empty example tables, adding fields, and so on. We do not discuss these features but assume that they are available.

Sailors	*sid*	*sname*	*rating*	*age*
I.	74	Janice	7	41

We can insert several tuples, computed essentially through a query, into the Sailors relation as follows:

Sailors	*sid*	*sname*	*rating*	*age*
I.	_Id	_N		_A

Students	*sid*	*name*	*login*	*age*	*Conditions*
	_Id	_N		_A	_A > 18 OR _N LIKE 'C%'

We insert one tuple for each student older than 18 or with a name that begins with C. (QBE's LIKE operator is similar to the SQL version.) The *rating* field of every inserted tuple contains a *null* value. The following query is very similar to the previous query, but differs in a subtle way:

Sailors	*sid*	*sname*	*rating*	*age*
I.	_Id1	_N1		_A1
I.	_Id2	_N2		_A2

Students	*sid*	*name*	*login*	*age*
	_Id1	_N1		_A1 > 18
	_Id2	_N2 LIKE 'C%'		_A2

The difference is that a student older than 18 with a name that begins with 'C' is now inserted *twice* into Sailors. (The second insertion will be rejected by the integrity constraint enforcement mechanism because *sid* is a key for Sailors. However, if this integrity constraint is not declared, we would find two copies of such a student in the Sailors relation.)

We can delete all tuples with $rating > 5$ from the Sailors relation as follows:

Sailors	*sid*	*sname*	*rating*	*age*
D.			> 5	

We can delete all reservations for sailors with $rating < 4$ by using:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id		< 4	

Reserves	*sid*	*bid*	*day*
D.	_Id		

We can update the age of the sailor with *sid* 74 to be 42 years by using:

Sailors	*sid*	*sname*	*rating*	*age*
	74			U.42

The fact that *sid* is the key is significant here; we cannot update the key field, but we can use it to identify the tuple to be modified (in other fields). We can also change the age of sailor 74 from 41 to 42 by incrementing the age value:

Sailors	*sid*	*sname*	*rating*	*age*
	74			U._A+1

6.8.1 Restrictions on Update Commands

There are some restrictions on the use of the I., D., and U. commands. First, we cannot mix these operators in a single example table (or combine them with P.). Second, we cannot specify I., D., or U. in an example table that contains G. Third, we cannot insert, update, or modify tuples based on values in fields of other tuples in the same table. Thus, the following update is incorrect:

Sailors	*sid*	*sname*	*rating*	*age*
		john		U._A+1
		joe		_A

This update seeks to change John's age based on Joe's age. Since *sname* is not a key, the meaning of such a query is ambiguous—should we update *every* John's age, and if so, based on *which* Joe's age? QBE avoids such anomalies using a rather broad restriction. For example, if *sname* were a key, this would be a reasonable request, even though it is disallowed.

6.9 DIVISION AND RELATIONAL COMPLETENESS *

In Section 6.6 we saw how division can be expressed in QBE using COUNT. It is instructive to consider how division can be expressed in QBE without the use of aggregate operators. If we don't use aggregate operators, we cannot express division in QBE without using the update commands to create a temporary relation or view. However,

taking the update commands into account, QBE is relationally complete, even without the aggregate operators. Although we will not prove these claims, the example that we discuss below should bring out the underlying intuition.

We use the following query in our discussion of division:

Find sailors who have reserved all boats.

In Chapter 4 we saw that this query can be expressed in DRC as:

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \ \wedge \forall \langle B, BN, C\rangle \in Boats$$
$$(\exists \langle Ir, Br, D\rangle \in Reserves (I = Ir \wedge Br = B))\}$$

The $\forall$ quantifier is not available in QBE, so let us rewrite the above without $\forall$:

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge \neg\exists \langle B, BN, C\rangle \in Boats$$
$$(\neg\exists \langle Ir, Br, D\rangle \in Reserves (I = Ir \wedge Br = B))\}$$

This calculus query can be read as follows: "Find Sailors tuples (with *sid* I) for which there is no Boats tuple (with *bid* B) such that no Reserves tuple indicates that sailor I has reserved boat B." We might try to write this query in QBE as follows:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id	P._S		

Boats	*bid*	*bname*	*color*
¬	_B		

Reserves	*sid*	*bid*	*day*
¬	_Id	_B	

This query is illegal because the variable $\_B$ does not appear in any positive row. Going beyond this technical objection, this QBE query is ambiguous with respect to the *ordering* of the two uses of ¬. It could denote either the calculus query that we want to express or the following calculus query, which is not what we want:

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge \neg\exists \langle Ir, Br, D\rangle \in Reserves$$
$$(\neg\exists \langle B, BN, C\rangle \in Boats (I = Ir \wedge Br = B))\}$$

There is no mechanism in QBE to control the order in which the ¬ operations in a query are applied. (Incidentally, the above query finds all Sailors who have made reservations only for boats that exist in the Boats relation.)

One way to achieve such control is to break the query into several parts by using temporary relations or views. As we saw in Chapter 4, we can accomplish division in

two logical steps: first, identify *disqualified* candidates, and then remove this set from the set of all candidates. In the query at hand, we have to first identify the set of *sid*s (called, say, BadSids) of sailors who have not reserved some boat (i.e., for each such sailor, we can find a boat not reserved by that sailor), and then we have to remove BadSids from the set of *sid*s of all sailors. This process will identify the set of sailors who've reserved all boats. The view BadSids can be defined as follows:

Sailors	*sid*	*sname*	*rating*	*age*
	_Id			

Reserves	*sid*	*bid*	*day*
¬	_Id	_B	

Boats	*bid*	*bname*	*color*
	_B		

BadSids	*sid*
I.	_Id

Given the view BadSids, it is a simple matter to find sailors whose *sid*s are not in this view.

The ideas in this example can be extended to show that QBE is *relationally complete*.

6.10 POINTS TO REVIEW

- QBE is a user-friendly query language with a graphical interface. The interface depicts each relation in tabular form. **(Section 6.1)**
- Queries are posed by placing constants and variables into individual columns and thereby creating an example tuple of the query result. Simple conventions are used to express selections, projections, sorting, and duplicate elimination. **(Section 6.2)**
- Joins are accomplished in QBE by using the same variable in multiple locations. **(Section 6.3)**
- QBE provides a limited form of set difference through the use of ¬ in the relation-name column. **(Section 6.4)**
- Aggregation (`AVG.`, `COUNT.`, `MAX.`, `MIN.`, and `SUM.`) and grouping (`G.`) can be expressed by adding prefixes. **(Section 6.5)**
- The condition box provides a place for more complex query conditions, although queries involving `AND` or `OR` can be expressed without using the condition box. **(Section 6.6)**
- New, unnamed fields can be created to display information beyond fields retrieved from a relation. **(Section 6.7)**

- QBE provides support for insertion, deletion and updates of tuples. **(Section 6.8)**
- Using a temporary relation, division can be expressed in QBE without using aggregation. QBE is relationally complete, taking into account its querying and view creation features. **(Section 6.9)**

EXERCISES

Exercise 6.1 Consider the following relational schema. An employee can work in more than one department.

Emp(*eid:* `integer`, *ename:* `string`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`)
Dept(*did:* `integer`, *dname:* `string`, *managerid:* `integer`, *floornum:* `integer`)

Write the following queries in QBE. Be sure to underline your variables to distinguish them from your constants.

1. Print the names of all employees who work on the 10th floor and make less than $50,000.
2. Print the names of all managers who manage three or more departments on the same floor.
3. Print the names of all managers who manage 10 or more departments on the same floor.
4. Give every employee who works in the toy department a 10 percent raise.
5. Print the names of the departments that employee Santa works in.
6. Print the names and salaries of employees who work in both the toy department and the candy department.
7. Print the names of employees who earn a salary that is either less than $10,000 or more than $100,000.
8. Print all of the attributes for employees who work in some department that employee Santa also works in.
9. Fire Santa.
10. Print the names of employees who make more than $20,000 and work in either the video department or the toy department.
11. Print the names of all employees who work on the floor(s) where Jane Dodecahedron works.
12. Print the name of each employee who earns more than the manager of the department that he or she works in.
13. Print the name of each department that has a manager whose last name is Psmith and who is neither the highest-paid nor the lowest-paid employee in the department.

Exercise 6.2 Write the following queries in QBE, based on this schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *city:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Orders(*sid:* `integer`, *pid:* `integer`, *quantity:* `integer`)

1. For each supplier from whom all of the following things have been ordered in quantities of at least 150, print the name and city of the supplier: a blue gear, a red crankshaft, and a yellow bumper.
2. Print the names of the purple parts that have been ordered from suppliers located in Madison, Milwaukee, or Waukesha.
3. Print the names and cities of suppliers who have an order for more than 150 units of a yellow or purple part.
4. Print the *pid*s of parts that have been ordered from a supplier named American but have also been ordered from some supplier with a different name in a quantity that is greater than the American order by at least 100 units.
5. Print the names of the suppliers located in Madison. Could there be any duplicates in the answer?
6. Print all available information about suppliers that supply green parts.
7. For each order of a red part, print the quantity and the name of the part.
8. Print the names of the parts that come in both blue and green. (Assume that no two distinct parts can have the same name and color.)
9. Print (in ascending order alphabetically) the names of parts supplied both by a Madison supplier and by a Berkeley supplier.
10. Print the names of parts supplied by a Madison supplier, but not supplied by any Berkeley supplier. Could there be any duplicates in the answer?
11. Print the total number of orders.
12. Print the largest quantity per order for each *sid* such that the minimum quantity per order for that supplier is greater than 100.
13. Print the average quantity per order of red parts.
14. Can you write this query in QBE? If so, how?
Print the sid*s of suppliers from whom every part has been ordered.*

Exercise 6.3 Answer the following questions:

1. Describe the various uses for unnamed columns in QBE.
2. Describe the various uses for a conditions box in QBE.
3. What is unusual about the treatment of duplicates in QBE?
4. Is QBE based upon relational algebra, tuple relational calculus, or domain relational calculus? Explain briefly.
5. Is QBE relationally complete? Explain briefly.
6. What restrictions does QBE place on update commands?

PROJECT-BASED EXERCISES

Exercise 6.4 Minibase's version of QBE, called MiniQBE, tries to preserve the spirit of QBE but cheats occasionally. Try the queries shown in this chapter and in the exercises, and identify the ways in which MiniQBE differs from QBE. For each QBE query you try in MiniQBE, examine the SQL query that it is translated into by MiniQBE.

BIBLIOGRAPHIC NOTES

The QBE project was led by Moshe Zloof [702] and resulted in the first visual database query language, whose influence is seen today in products such as Borland's Paradox and, to a lesser extent, Microsoft's Access. QBE was also one of the first relational query languages to support the computation of transitive closure, through a special operator, anticipating much subsequent research into extensions of relational query languages to support recursive queries. A successor called Office-by-Example [701] sought to extend the QBE visual interaction paradigm to applications such as electronic mail integrated with database access. Klug presented a version of QBE that dealt with aggregate queries in [377].