5.9 A Critique: SQL, QUEL

SQL and QUEL are easier to use and more powerful as data sublanguages than the ones used in DBMSs based on the network and hierarchical models. However, these languages do not fully support some of the basic features of the relational data model: the concept of domains, entity and referential integrity and hence the concept of primary and foreign keys. Furthermore, these languages are redundant in the sense that the same query may be expressed in more than one way.

Redundancy is not a sin as long as different ways of expressing the same query yield the same results in approximately the same period of time. However, tests with a number of implementations of SQL, the most widely available query language for relational DBMSs, indicate a wide variation in response time. Furthermore, some forms of the query generate duplicate tuples whereas others do not.

Proponents of QUEL claim that it is more orthogonal and powerful than SQL. The term **orthogonal** is used in programming languages to mean that concepts and constructs are designed independently and can be used consistently in a uniform manner. In an orthogonal language, there are no special cases and few restrictions imposed on the use of the components of the language. The current SQL standard is viewed as one that tried to reconcile the various commercial implementations and came up with one that is, in effect, the lowest common denominator. An attempt is currently underway to upgrade the SQL standard.

The following illustrates the nonorthogonality of SQL. The first version is valid while the second, though symmetrical, is invalid. This is so because the nested select operand is required to be on the right-hand side of the θ operator.

First version:

select *Name*
from EMPLOYEE
where *Pay_Rate* >
  (**select avg** *(Pay_Rate)*
  from EMPLOYEE)

Second version:

select *Name*
from EMPLOYEE
where (**select avg** *(Pay_Rate)*
  from EMPLOYEE) ≤ *Pay_Rate*

As mentioned earlier, the select statement of SQL represents the following relational algebraic operations:

projection$_{\text{(represented by the target list)}}$ (selection $_{\text{(represented by the where clause)}}$ (cartesian product of the relations represented by the from list))

It is not possible to change the order of these operations in SQL. Consequently, the user has to express a query in this format, making the query less like a natural language query.

The treatment of nested select statements in various set operators such as exists, θany, θall, in, and contains is also nonuniform. Whereas a nested select statement

producing a relation as the result is required in the case of exists, nested select is only permitted if the value produced in the case of one of the operators {=, ≠, >, ≥, <, ≤} is a relation of cardinality and degree one (a single value). On the other hand, the result of the nested select in the case of one of the set operators {**θany, θall, in, contains**} is required to be a relation of degree one and arbitrary cardinality.

Suppose we want to create a table that contains the names of employees, their pay rate, and, for comparison, the average pay rate. This can be expressed in QUEL as shown in Example 5.42c. However, an attempt to create such a table using the following SQL statement, though intuitively valid, will fail because such usage is illegal in SQL. The reason is that the select is a projection and the cardinality of *Name, Pay_Rate,* is not the same as the cardinality of **avg***(Pay_Rate).*

select *Name, Pay_Rate,* **avg***(Pay_Rate)*
from EMPLOYEE

However, the following is legal and produces a table of skill and the average pay rate for each skill:

select *Skill,* **avg***(Pay_Rate)*
from EMPLOYEE
group by *Skill*

QUEL allows updates to involve values from two relations. As such, the pay rates of employee in the relation EMPLOYEE can be adjusted according to the values in a relation ADJUSTMENT shown below:

range of a **is** ADJUSTMENT
range of e **is** EMPLOYEE
replace (e.*Pay_Rate* = a.*Raise* * e.*Pay_Rate*)
where e.*Skill* = a.*Skill*

ADJUSTMENT

Skill	*Raise*
waiter	1.08
bartender	1.07
busboy	1.12
hostess	1.09
maître d'	1.08
chef	1.09

A similar attempt to use a value from another relation, as illustrated below, is invalid in SQL:

update EMPLOYEE
set *Pay_Rate* = *Pay_Rate* ***(select** a.*Raise*
from ADJUSTMENT a
where EMPLOYEE.*Skill* = a.*Skill)*

However, in some implementations of SQL the following statement would produce the required adjustment in *Pay_Rates.* It should be obvious that for this state-

ment to work correctly, the relation ADJUSTMENT must have a tuple corresponding to each value of *Skill* in EMPLOYEE.

```
update EMPLOYEE
set Pay_Rate = (select Pay_Rate * a.Raise
                from ADJUSTMENT a
                where EMPLOYEE.Skill = a.Skill)
```

The nonorthogonality of SQL in allowing nested query in some places and not in others is illustrated below. Whereas the select statement on the left is legal in SQL a similar form in the update statement on the right is not valid in all implementations of SQL.

```
select Name
from EMPLOYEE
where Empl_No =
      (select Empl_No
       from DUTY_ALLOCATION
       where Shift = 3)
```

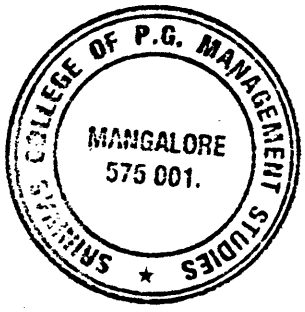
```
update EMPLOYEE
set Pay_Rate = 1.3 * Pay_Rate
where Empl_No in
      (select Empl_No
       from DUTY_ALLOCATION
       where Shift = 3)
```

QUEL, on the other hand, has required the use of tuple variables in its query to date. This restriction has been modified and QUEL now allows the use of a relation name as the tuple variable. This was implemented by a query modification introducing the relation name as a tuple variable. However, as illustrated in the following query, using both a tuple variable and a relation name could produce an incorrect result:

```
range of e is EMPLOYEE
replace EMPLOYEE(Pay_Rate = 10.50)
where e.Empl_No = 123456
```

This query is modified by the introduction of a range statement:

```
range of e is EMPLOYEE
range of EMPLOYEE is EMPLOYEE
replace EMPLOYEE(Pay_Rate = 10.50)
where e.Empl_No = 123456
```



The result is unexpected since the query sets the pay rate of *all* employees to 10.50 if there exists an employee with the number 123456.

One of the more mystifying features of QUEL is the scope rule of tuple variables in aggregation operators and aggregate functions. In aggregation operators tuple variables are strictly local, whereas in aggregate functions the presence of the by clause requires that the tuple variable used in that clause has a global scope. Consider the query "Find the average *Pay_Rate* by *Skill*." The QUEL version of this query is shown in Example 5.45c. However, it may be expressed by a novice user using an additional tuple variable as follows:

```
range of e is EMPLOYEE
range of el is EMPLOYEE
retrieve (e.Skill, Avg_Rate = avg(el.Pay_Rate by e.Skill))
```

This query shows the global scope of the tuple variable used in the by clause, which is the same as that used outside the aggregate function. The tuple variable el is

average pay rate, the result is derived as 0.0. The result relation produced by this query is evidently wrong. This would be not apparent to the user unless he or she had known the contents of the MORE_EMPLOYEE relation and had computed some sample results.

Let us modify the query as shown below. Here the by clause forces the tuple variable in both the aggregate functions to be global.

```
range of e is MORE_EMPLOYEE
range of el is MORE_EMPLOYEE
retrieve (e.Skill, Low_Avg_Rate =
    avg(e.Pay_Rate by e.Skill where e.Pay_Rate <
    avg(el.Pay_Rate by e.Skill where el.Skill = e.Skill) ) )
```

The second average is now computed using only those tuples of the join of MORE_EMPLOYEE with itself where the skill is the same as one outside the function. This indicates the correct tuples to choose for computing the low average pay rate. The result is shown in Figure 5.8.

We can simplify the last query as shown below. This simplified query gives the same result as shown in Figure 5.8.

```
range of e is MORE_EMPLOYEE
retrieve (e.Skill, Low_Avg_Rate =
            avg(e.Pay_Rate by e.Skill where e.Pay_Rate <
            avg(e.Pay_Rate by e.Skill) ) )
```

As illustrated above, a mixture of local and global scope of tuple variables in QUEL tends to create confusion and retrieve incorrect data.

The SQL version of this query is relatively simple as shown below:

```
select e.Skill, avg(e.Pay_Rate)
from MORE_EMPLOYEE e
where e.Pay_Rate < (select avg(el.Pay_Rate)
                    from MORE_EMPLOYEE el
                    where el.Skill = e.Skill)
group by e.Skill
```

Figure 5.8 Correct values by *Skill* of average *Pay_Rate* of employees below the average for their skills.

Skill	*Low Avg_Rate*
bartender	8.59
bellboy	4.50
busboy	4.50
chef	14.00
hostess	4.80
maître d'	8.00
waiter	6.88
chef	14.00

The above discussion illustrates that neither SQL nor QUEL are perfect for expressing all queries. A user has to know the "correct" versions without which the information gleaned from the DBMS may be incorrect. The user may have no way of ascertaining the correctness of the response.

The SQL standard is under review and as with all such standards will go through a number of versions. It is hoped that future standards will address some of the criticisms leveled at SQL.

5.10 QBE

Query-By-Example (QBE) was originally developed by M. M. Zloof at IBM's Yorktown Heights Research Laboratory and has now been marketed for various relational systems from IBM as part of their QMF (Query Management Facility). In QMF, QBE is implemented not as in the system developed by Zloof, but rather by translating the QBE queries into equivalent SQL queries. Other relational DBMSs such as DBASE IV, INGRES, and ORACLE have some form of example or form-based query system.

QBE is based on domain calculus and has a two-dimensional syntax. The queries are written in the horizontal and vertical dimensions of a table. Queries are formed by entering an example of a possible answer in a **skeleton** (empty) **table,** as shown in Figure 5.9. This example contains variables as in domain calculus and specifies the conditions that have to be satisfied by the response. Conditions specified on a single row of the table are generally considered to be **conjunctive** (i.e., "*and*ed"); conditions entered on separate rows are **disjunctive** (i.e., "*or*ed"). An empty skeleton is displayed by pressing a function key.

The skeleton table does not have column headings. The first column is used for the relation name.

To get a list of relations, we enter P. for the PRINT command in the first column of the column heading:

P.				

To get the attribute names for a given relation we enter the relation name followed by a P.

DUTY_ALLOCATION P.				

The fact that all details are required is indicated by the P. under the relation name.

$$\{p,e,s,d \mid <p,e,s,d> \in \text{DUTY\_ALLOCATION} \wedge e = 123456\}$$

DUTY_ALLOCATION	*Posting_No*	*Empl_No*	*Shift*	*Day*
P.		123456		

■

QBE supports the usual comparison operators: $=$, $\neq$(not equal), $<$, $\leq$, $>$, $\geq$; $=$ is normally omitted as seen in the previous example. The Boolean operators and, or, not are also supported. Conditions specified within a single row are *and*ed. For multiple conditions on the same column, k, to be *and*ed, QBE requires multiple rows with the same example element in the *k*th column of each row. To specify conditions to be *or*ed we use different rows with different example elements.

Example 5.56

"Get names of employees with the skill of chef earning more than $14.00 per hour." The above query reads "Get employee names where *Skill* = 'chef' and *Pay_Rate* > 14.00" and is the domain calculus query:

$$\{n \mid <e,n,s,p> \in \text{EMPLOYEE} \wedge s = \text{'chef'} \wedge p > 14.00\}$$

This query requires two conditions to be true for the tuples that are retrieved. It can be expressed on the skeleton table as illustrated below:

EMPLOYEE	*Name*	*Skill*	*Pay_Rate*
	P.EX	'chef'	>14.00

■

In the above example not all attribute names of the employee relation were listed. It is possible in QBE to eliminate columns from the display if they are irrelevant to the query.

Example 5.57

"Get names of chefs who earn more than $10 per hour but less than $20 per hour." To specify a conjunctive predicate of the form $P_1(attr_i) \wedge P_2(attr_i) \wedge \ldots P_n(attr_i)$, QBE allows multiple columns for the same attribute in the skeleton table. Hence this query can be expressed as shown below:

EMPLOYEE	*Empl_No*	*Name*	*Skill*	*Pay_Rate*	*Pay_Rate*
		P.EX	chef	>10.00	<20.00

An alternate scheme with multiple rows with the same domain variable to express the conjunctive predicate can also be used. The query could be reexpressed as "Get employee names whose *Skill* = 'chef' with *Pay_Rate*>10 AND (the same) employee names whose *Skill* is also 'chef' with *Pay_Rate*<20." This is expressed in QBE by two rows with the same variable in the *Name* column as indicated below:

EMPLOYEE	*Empl_No*	*Name*	*Skill*	*Pay_Rate*
		P.EX	chef	>10.00
		EX	chef	<20.00

The following example illustrates a disjunctive predicate.

Example 5.58

"Get names of employees who are either chefs or earn more than $8 per hour." In this query, the conditions to be *or*ed are indicated by using two rows in the skeleton table with different variable names for the *Name* column.

EMPLOYEE	*Empl_No*	*Name*	*Skill*	*Pay_Rate*
		P.EX	chef	
		P.EY		>8.00

■

Data from multiple tables can be manipulated as shown in Example 5.59.

Example 5.59

"Get shift details for the employee named Ian." This query is "Print *Posting No, Shift* and *Day* (e.g., P1,S1,D1 respectively) for employee number EX where EX is the *Empl_No* for employee Ian. The response to the query involves a join of relations EMPLOYEE and DUTY_ALLOCATION. In QBE the join is implemented by utilizing the example element EX as a link between these relations. The link in QBE is used whenever a join would be used in relational algebra.

DUTY_ALLOCATION	*Posting_No*	*Empl_No*	*Shift*	*Day*
P.	P.P1	EX	P.S1	P.D1

EMPLOYEE	*Empl_No*	*Name*	*Skill*	*Pay_Rate*
		P.EX		P.RX P.AVG.ALL.RY

(d) "Find names of employees with *Pay_Rate* less than the average *Pay_Rate*."

	P.EX	P.RX AVG.ALL.RY

CONDITIONS
RX < AVG.ALL.RY

■

5.10.3 Categorization in QBE

The equivalent of the SQL group by operator is obtained in QBE by preceding the variable with G.

Example 5.62

(a) "Get count of employees on each shift."

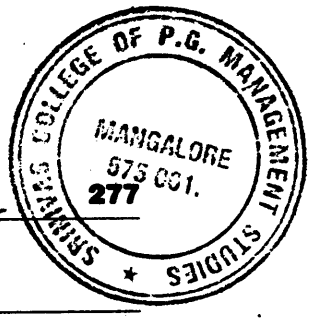
DUTY_ALLOCATION	*Empl_No*	*Shift*	*Day*
	P.CNT.ALL.EX	G.SX	

(b) "Get employee numbers of all employees assigned a duty on dates in addition to the date 19860419."

DUTY_ALLOCATION	*Empl_No*	*Day*
	EX P.G.EX	19860419

CONDITIONS
CNT.ALL.EX > 1

■



5.10.4 Updates

QBE includes the three update operations for inserting, modifying, and deleting. These are indicated on the skeleton table in the relation name column by I. (insert), U. (modify/replace), and D. (delete). For the U. update operation based on an old value, the user first specifies the old version and next the new version. We illustrate the syntax for specifying these operations in the following examples.

Example 5.63

(a) "Insert a record into DUTY_ALLOCATION at *Posting_No* 321 for *Empl No* 123458, *Shift* 2, and *Day* 19860421."

DUTY_ALLOCATION	*Posting_No*	*Empl_No*	*Shift*	*Day*
I.	321	123458	2	19860421

Here the I. in the relation name column indicates the insertion operation. The values for the columns are indicated on the skeleton of the table.

(b) "Copy DUTY_ALLOCATION into NEW_DUTY_ALLOCATION."

DUTY_ALLOCATION	*Posting_No*	*Empl_No*	*Shift*	*Day*
	PX	EX	SX	DX

NEW_DUTY_ALLOCATION	*Posting_No*	*Empl_No*	*Shift*	*Day*
I.	PX	EX	SX	DX

■

Here the I. in the relation name column for the NEW_DUTY_ALLOCATION table indicates the insertion operation. The similarly named variables in DUTY_ALLOCATION and NEW_DUTY_ALLOCATION indicate the source of the values to be used for the insertion.

(c) "Copy into NEW_DUTY_ALLOCATION records for Shift 1 in DUTY _ALLOCATION."

DUTY_ALLOCATION	*Posting_No*	*Empl_No*	*Shift*	*Day*
D.		EX		

In the first line of the EMPLOYEE skeleton we indicate that we are interested in the employee with the name of Ian and hence select these tuples. On the second line we indicate that these tuples are to be deleted. The use of the EX with D. in the DUTY_ALLOCATION skeleton indicates that tuples satisfying this predicate are to be deleted as well. ■

5.11 Concluding Remarks

In this chapter we considered some of the salient features of the more popular commercial data manipulation languages. We can see how they borrow heavily from relational algebra and calculus concepts. In query design, relatively little attention needs to be paid to evaluation. Users benefit greatly from this philosophy. In some ways data manipulation resembles programming and, like good programming, comes from practice. The requirement is that we be able to express exactly what we desire.

We can reflect on the complexity of what is achieved by some very simple queries. As is normal in most database systems, suppose that every relation is supported by an underlying file of records. Let us consider the SQL query

```
select R.A, S.D
from R, S
where R.B = S.C
```

Let the tuples of relations R and S be stored as records in the files FR and FS, respectively. The above query requires that starting with the first record of FR (tuple of R), we compare its field, B, with field C of every record of file FS, outputting field A value from FR and field D value from FS whenever the comparands are equal. For n records in file FR and m in file FS, this would require some m * n combinations. Even for moderate-sized relations this signifies a large number. In Chapter 10 we consider how we can optimize this query. More immediately, however, we should reflect on how to program this task in a file environment. In this case, the task of translating the query into a file processing program is easy. For more complex queries, the programming task is much more difficult. We can therefore appreciate the productivity improvements, among other benefits, of using a relational database system.

5.12 Summary

In this chapter we examined the commercial versions of languages used for relational database systems. These languages, unlike their theoretical counterparts, include facilities to define data as well as manipulate it.