The aggregation operators supported are any, avg, min, max, count, and sum, similar to the corresponding functions available in SQL. The operators **avg, count,** and **sum** have versions that eliminate duplicates before applying the operator. These "unique" versions are distinguished by the suffix u. The **any** aggregate operator can be used to check if any tuple satisfies a given qualification. The value returned by the **any** operator is 1 if the qualification is satisfied and 0 otherwise. The advantage of using the **any** operator as opposed to using the count operator is that if the qualification is satisfied, the processing of additional tuples is discontinued, resulting in a faster evaluation of the query. The format for using these operators is:

aggregation operator (<expression>)

The tuple variables appearing as arguments of an aggregate operator are always local to it and distinct from any tuple variable with the same name appearing external to the arguments of the aggregate operator. The aggregate operator could logically be considered to be processed separately, and a computed single value replaces it. We illustrate the use of some of these operators in the following examples.

Example 5.42

(a) "Obtain the average dish price."

range of r **is** MENU
retrieve *(Ave_Price* = **avg**(r.*Price))*

The term **avg**(r.*Price*) returns the average of the r.*Price* values. For our sample database the *Ave_Price* is 10.90.

(b) "Get minimum and maximum dish prices."

range of r **is** MENU
retrieve *(Minprice* = **min**(r.*Price*),
Maxprice = **max**(r.*Price*))

(c) "Get the average rate of pay for all employees and list it against each employees' names and rates of pay."

range of e **is** EMPLOYEE
retrieve (e.*Name*, e.*Pay_Rate*, *Avg_Pay* = **avg** (e.*Pay_Rate*))

The result of this query for our sample database is shown below:

Name	*Pay_Rate*	*Avg_Pay*
Ron	7.50	8.51
Jon	8.79	8.51
Don	4.70	8.51
Pam	4.90	8.51
Pat	4.70	8.51
Ian	9.00	8.51
Pierre	14.00	8.51
Julie	14.50	8.51

■

Note that in the query in Example 5.42c the aggregation operation is independent of the current tuple values. The average rate of pay from all employee tuples is returned by the avg operator. We see this important difference in the next few queries where the aggregates are themselves qualified.

Example 5.43

"Find the average rate of pay for employees with the skill of chef."

First attempt:

```
range of e is EMPLOYEE
retrieve (e.Empl_No, e.Skill, Avgchef_Pay =
    avg(e.Pay_Rate where e.Skill = 'chef'))
```

The result relation includes tuples with the above details for all employees including those who are not chefs. In the above query the qualification "e.*Skill* = 'chef'" applies only to the aggregate, not to the query. The aggregate qualification is local; it is not affected by and does not affect the rest of the query. Thus, the scheme of the result is *(Empl_No, Skill, Avgchef_Pay)*, and each tuple of the result relation contains the same value for the *Avgchef_Pay* attribute.

Second attempt: The query

```
range of e is EMPLOYEE
retrieve (e.Empl_No, e.Skill,
                Avgchef_Pay = avg(e.Pay_Rate))
where e.Skill = 'chef'
```

gets employee number and skill for all employees who are chefs and the average rate of pay of all employees (not just chefs).

The correct query (to get the employee number, skill, and average salaries of employees who are chefs) should be formulated as given below in the third attempt. Here we are using two qualification clauses; one is for the computation of the average salary of employees with a skill of chef and the other is to ensure that the result contains only tuples for chefs.

Third attempt:

```
range of e is EMPLOYEE
retrieve (e.Empl_No, e.Skill, Avgchef_Pay =
    avg(e.Pay_Rate where e.Skill = 'chef'))
    where e.Skill = 'chef'
```

■

The use of count operator is illustrated in Example 5.44.

Example 5.44

"Get the total number of employees."

```
range of e is EMPLOYEE
retrieve (cnt = count(e.Empl_No))
```

Because we defined *Empl_No* as the key for the relation EMPLOYEE we expect no duplicate employee records and the unique version of count is unnecessary. ■

Another aggregation facility supported in QUEL is called the **aggregate function.** This facility allows data to be grouped into categories and aggregations to be performed separately on each group. The aggregate function is invoked by including the **by** clause in the expression for the aggregate operator:

by <by-list>

Unlike simple aggregates, aggregate functions are not local; the by-list links the function to the rest of the query. The tuple variable appearing in by-list is global to the query and is therefore restricted by the qualification of the entire query as well as by any aggregate qualification. The value of an aggregate function is a set of values.

The aggregate function any can be used as an existential quantifier. The use of it in any(. . .) = 1 or any(. . .) = 0 makes the quantification explicit, as illustrated in Example 5.45e.

Example 5.45

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

range of e **is** DUTY_ALLOCATION
retrieve(cnt = **count**(e.*Empl_No* **by** e.*Shift*))

(b) "Find the number of employees on shift number 1."

range of e **is** DUTY_ALLOCATION
retrieve (cnt = **count** (e.*Empl_No* **by** e.*Shift*))
where e.*Shift* = 1

cnt
4

The tuple variable e is global and the by clause links it to the **where** clause, limiting the count to those for shift number 1. The result of this query for the sample database given in Figure 5.4 is as shown above.

A simpler formulation of this query, where the use of a local tuple variable is acceptable, is given below:

range of e **is** DUTY_ALLOCATION
retrieve (cnt = **count** (e.*Empl_No* **where** e.*Shift* = 1))

(c) "Determine the average *Pay_Rate* by skill."

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

Skill	*Avg_Rate*
waiter	7.50
bartender	8.79
busboy	4.70
hostess	4.90
bellboy	4.70
maître d'	9.00
chef	14.25

The query shows the global scope of the tuple variable used in the by clause. Here the use of the by clause causes the tuple variable associated with it to be global; it is the same as the one used outside the aggregate function. The tuple variable associated with e.*Pay_Rate* is strictly local. The avg function generates a number of values of average pay rate, namely one for each skill. However, a skill and its corresponding value is displayed only once, as shown above for the sample EMPLOYEE relation in Figure 5.4.

(d) "Obtain the average of the total pay rate for each skill."

range of e **is** EMPLOYEE
retrieve (*Avg_of_Total* = **avg**(**sum** (e.*Pay_Rate*
by e.*Skill*)))

The above query demonstrates the aggregate function nested in an aggregate operator. The sum aggregate function generates the sum of *Pay_Rates* by *Skill* giving the set {7.50, 8.79, 4.70, 4.90, 4.70, 9.00, 28.50} as its result for the sample EMPLOYEE relation of Figure 5.4.

The avg operator is applied to this set to get a single value, indicated below:

Avg_of_Total
9.73

Note that this query is not the same as the following, which generates the value 8.51, being the overall average value of the *Pay_Rate* for all employees:

retrieve(*Overall_Avg_Rate* = **avg**(EMPLOYEE.*Pay_Rate*))

(e) "Get the names of employees who are assigned to *Posting_No* 321."

range of e **is** EMPLOYEE
range of d **is** DUTY_ALLOCATION
retrieve unique (e.*Name*)
where any (d.*Empl_No* **by** e.*Empl_No*
where d.*Posting_No* = 321
and d.*Empl_No* = e.*Empl_No*) = 1

In this example, the any aggregate function is evaluated over the argument attribute *Empl_No*, which is grouped using the by clause. The predicates specified by the where clause must be satisfied by each value of the argument. For our sample database, the result of the query is the employee names Ian and Ron.

The following can be used to find the names of employees who are not assigned to *Posting_No* 321:

range of e **is** EMPLOYEE
range of d **is** DUTY_ALLOCATION
retrieve unique (e.*Name*)

where any (d.*Empl_No* **by** e.*Empl_No*
where d.*Posting_No* = 321
and d.*Empl_No* = e.*Empl_No*) = 0

For our sample database, the result of the query is the employee names Don, Jon, Julie, Pam, Pat, Pierre. Note that the function count could have been used here instead of any giving the same result.

(f) "Get the *Empl_No* of the employees who are assigned a duty on at least one date in addition to 19860419." The first version for this query uses the count operator and accesses each tuple of the relation. The second version, which uses the any operator, will terminate the evaluation of the where clause when it accesses the first tuple satisfying the qualification. The result in each case is the employee numbers 123461 and 123471.

First version:

range of d **is** DUTY_ALLOCATION
retrieve (d.*Empl_No*)
where d.*Day* = 19860419
and count(d.*Day* **by** d.*Empl_No*) > 1

Second version:

range of d **is** DUTY_ALLOCATION
retrieve (d.*Empl_No*)
where d.*Day* = 19860419
and any(d.*Day* **by** d.*Empl_No* **where** d.*Day* ≠ 19860419) = 1 ■

5.7.7 Retrieve into Temporary Relation

So far we have not considered what happens to the retrieved data; in an interactive environment the results would have been listed on the user's output device. It is also possible to assign the result of the retrieval to a relation. The format of such a retrieve command is:

retrieve into <new-relation > (<target list>)
[where <condition>]

The new relation will be created with the correct attribute names and the result of the query put into this relation. The content of the new relation will be similar to a simple retrieve statement.

This scheme of using a relation to accept the result of a retrieve statement can be used in places where SQL uses a nested subquery, as illustrated in the next example.

Example 5.46

"Get total amount for Bill table 12 for the date 19860419." Here we create a temporary relation ITEMIZED_BILL and subsequently use it to find the total amount for the bill.

```
range of b is BILL
range of m is MENU
range of o is ORDR
retrieve into ITEMIZED_BILL(b.Bill#,m.Description,m.Price,
            o.Qty, Dish_Total = m.Price*o.Qty)
where b.Table# = 12
   and b.Day = 19860419
   and o.Dish# = m.Dish#
   and b.Bill# = o.Bill#
range of i is ITEMIZED_BILL
retrieve unique(i.Bill#, Total_Amount = sum(i.Dish_Total)) ■
```

5.7.8 Updates

So far we have seen the QUEL data retrieval commands. Data in relations can also be changed using the three update commands append, replace, and delete. The format of the append command is:

append to <relation name> (<value list>)
[where <condition>]

and the value list takes the form

<value list> ·· = <attribute name> = <value expression> [,<value list>]

Append is used to insert new tuples into a relation. The **replace** and **delete** commands are used to replace or delete existing tuples. Thus the append requires the use of a relation name and the replace and delete commands should use a tuple variable. The format of the replace and delete commands is:

replace <tuple variable> (<value list>)
[where <condition>]

delete <tuple variable>
[where <condition>]

Example 5.47

(a) "Append a tuple to DUTY_ALLOCATION for *Posting_No* = 322, *Empl_No* = 123457, *Shift* = 2, *Day* = 19860421."

```
append to DUTY_ALLOCATION
(Posting_No = 322, Empl_No = 123457, Shift = 2,
Day = 19860421)
```

(b) "Copy the DUTY_ALLOCATION relation into NEW_DUTY_ALLOCATION."

```
range of d is DUTY_ALLOCATION
append to NEW_DUTY_ALLOCATION (d.all)
```

In this example, all tuples from the DUTY_ALLOCATION relation are copied into NEW_DUTY_ALLOCATION.

(c) "Copy only the tuples for shift 1 into the NEW_DUTY_ALLOCATION."

```
range of d is DUTY_ALLOCATION
    append to NEW_DUTY_ALLOCATION (d.all)
    where d.Shift = 1 ■
```

Example 5.48 illustrates the use of the replace command.

Example 5.48

(a) "Increase the pay rate of all employees by 10%."

```
range of e is EMPLOYEE
replace e (Pay_Rate = 1.1 * e.Pay_Rate)
```

The value for the attribute *Pay_Rate* in each tuple is increased by 10%. The other attributes are unchanged.

(b) "Increase the pay rate of all waiters by 10%."

```
range of e is EMPLOYEE
replace e (Pay_Rate = 1.1 * e.Pay_Rate)
    where e.Skill = 'waiter'
```

(c) To insert the total amount and the suggested tip into BILL with Bill# = 9234 from the relation ITEMIZED_BILL, we can use the following statements:

```
range of i is ITEMIZED_BILL
range of b is BILL
replace b
 (Total = sum(i.Dish_Total where i.Bill# = 9234),
 Tip = 0.15*sum(i.Dish_Total where i.Bill# = 9234))
  where b.Bill# = 9234 ■
```

Example 5.49 illustrates the delete command.

Example 5.49

"Remove the record for employee with *Empl_No* 123457."

```
range of e is EMPLOYEE
delete e
where e.Empl_No = 123457
```

and to delete all tuples from a relation:

```
range of e is EMPLOYEE
delete e
```

The result of the last command is an empty relation. ■

```
retrieve (Emp_No = e.Empl_No,
          Emp_Profession = e.Skill)
where e.Empl_No > 123300
      and e.Empl_No < 123460 ■
```

Such query modifications produce an appropriate external scheme to conceptual scheme mapping in a orderly manner. Updates via view, create problems similar to the ones we discussed under SQL.

Once defined, a view can be used until it is destroyed by means of a destroy statement as follows:

destroy EMP_VIEW

5.7.10 Remarks

Other QUEL commands deal with database creation, database removal, interface to the file system, index organization, and index modification. These do not deal specifically with data manipulation, so we have not emphasized them here.

The commercial version of INGRES provides a form-based interface, a report writer, interactive as well as embedded SQL and QUEL with HLL interface to BASIC, C, COBOL, Pascal, and PL/I. The database response has been much improved (about one order) over the INGRES used in the academic milieu.

5.8 Embedded Data Manipulation Language

SQL and QUEL only provide facilities to define and retrieve data interactively. To extend the data manipulation operations of these languages, for example to separately process each tuple of a relation, these languages have to be used with a traditional high-level language (HLL). Such a language is called a **host language** and the program is called the **host program.** The use of a database system in applications written in an HLL requires that the DML statements be embedded in the host programs. All the statements and features that are available to an interactive user must be available to the application programmer using the HLL. The DML statements are distinguished by means of a special symbol or are invoked by means of a subroutine call.

One approach that is commonly used is to mark the DML statements and partially parse them during a precompilation step to look for statements and variables from the host HLL appearing in DML statements. Such variables are appropriately identified by looking for a variable declaration in the host program or by appropriately marking such variables (e.g., with a colon). In this way, it is possible to use identical names for both the HLL variables and the objects in the database.

The need for domain compatibility between host language variables and constants and database attributes has to be observed in the design and writing of HLL programs with embedded DML statements. Any data type mismatch between HLL variables and DML attributes must be resolved. One way to handle type mismatch is to do type conversion at run time. Such type conversions must either be established