Example 5.18

(a) "Find the names and the rate of pay of all employees who are not allocated a duty." This query can be expressed in SQL using the not exists set operator.

```
select Name, Pay_Rate
from EMPLOYEE
where not exists
(select *
from DUTY_ALLOCATION
where EMPLOYEE.Empl_No =
   DUTY_ALLOCATION.Empl_No)
```

Name	*Pay_Rate*
Pam	4.90
Pat	4.70
Julie	14.50

In this example, for each employee tuple from the EMPLOYEE relation, the not exists clause is evaluated. If there is at least one tuple in DUTY_ALLOCATION for that employee, the second select statement produces a nonempty result whereby the not exists expression evaluates to the false value. The tuple is not included in the result, which is shown above.

The query can also be expressed using not in, as illustrated below:

```
select EMPLOYEE.Name, EMPLOYEE.Pay_Rate
from EMPLOYEE
where EMPLOYEE.Empl_No not in
      (select DUTY_ALLOCATION.Empl_No
      from DUTY_ALLOCATION)
```

(b) "Find the names of employees who are assigned to all positions that require a chef's skill." The tuple calculus expression for this query can be written as:

$$\{e[Name] | e \in EMPLOYEE \wedge \forall p(p \in POSITION \wedge p[Skill] = \text{'chef'} \rightarrow \exists d(d \in DUTY\_ALLOCATION \wedge d[Posting\_No] = p[Posting\_No] \wedge e[Empl\_No] = d[Empl\_No]))\}$$

Using $\forall x(f) \equiv \neg\exists x(\neg f)$, we can rewrite the tuple calculus expression as:

$$\{e[Name] | e \in EMPLOYEE \wedge \neg\exists p(\neg(p \in POSITION \wedge p[Skill] = \text{'chef'} \rightarrow \exists d(d \in DUTY\_ALLOCATION \wedge d[Posting\_No] = p[Posting\_No] \wedge e[Empl\_No]\ d[Empl\_No]))))\}$$

An alternate method of writing this query without the logical implication is to replace $f \rightarrow g$ by its equivalent form $\neg f \vee g$ to give the following expression:

$$\{e[Name] | e \in EMPLOYEE \wedge \neg\exists p(\neg(\neg(p \in POSITION \wedge p[Skill] = \text{'chef'}) \vee \exists d(d \in DUTY\_ALLOCATION \wedge d[Posting\_No] = p[Posting\_No] \wedge e[Empl\_No] = d[Empl\_No])))\}$$

which is equivalent to:

$$\{e[Name] | e \in \text{EMPLOYEE} \wedge \neg\exists p(p \in \text{POSITION} \wedge p[Skill] = \text{'chef'} \wedge \neg\exists d\ (d \in \text{DUTY\_ALLOCATION} \wedge d[Posting\_No] = p[Posting\_No] \wedge e[Empl\_No] = d[Empl\_No]))\}$$

This expression can be converted to SQL using not exists:

```
select e.Name
from EMPLOYEE e
where not exists
      (select p.Posting_No
      from POSITION p
      where p.Skill = 'chef'
      and not exists
      (select d.Empl_No
      from DUTY_ALLOCATION d
      where d.Posting_No = p.Posting_No
      and e.Empl_No = d.Empl_No ))
```

Here the first (outer) nested subquery finds the positions requiring a chef's skill. For each such position, the second (inner) nested subquery finds if the employee whose name is to be output is assigned to that position. If the result of the second nested subquery is empty (i.e., the employee being considered is not assigned to a position requiring the skill of a chef), the second not exists evaluates to true, causing the first not exists to evaluate to false, and the employee is not selected. In effect, we are saying that for those employees whose names are to be output, if there exists a position requiring a skill of chef, then there exists a tuple in DUTY_ALLOCATION where this position is assigned to that employee. If these combined tests evaluate to a true value, the attribute *Name* of the employee is output. For our sample database, the result of this query is (Pierre). We will get identical results even if a tuple such as (350, 123472, 19860420,1) were inserted in the DUTY_ALLOCATION relation. ■

Union

The traditional set theory union operator is **union.** Duplicates are removed from the result of a union.

Example 5.19

"Get employees who are waiters or work at *Posting_No 321*."

```
(select Empl_No
from EMPLOYEE
where Skill = 'waiter')
union
  (select Empl_No
  from DUTY_ALLOCATION
  where Posting_No = 321 ■
```

Empl_No
123456
123461

Minus

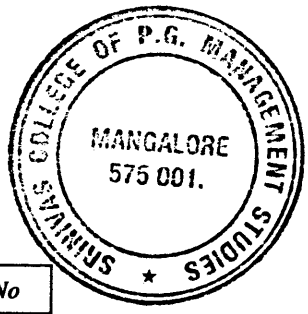
The traditional set theory difference operator is **minus.**

Example 5.20

(a) "Get employee numbers of persons who work at *Posting_No* 321, but don't have the skill of waiter". This query, which uses the minus operator, and its results are shown below:

```
(select distinct Empl_No
from DUTY_ALLOCATION
where Posting_No = 321)
    minus
      (select Empl_No
      from EMPLOYEE
      where Skill = 'waiter')
```

Empl_No
123461



(b) "Get a list of employees not assigned a duty."

```
(select Empl_No
 from EMPLOYEE )
 minus
      (select Empl_No
      from DUTY_ALLOCATION) ■
```

Empl_No
123459
123460
123472

Intersect

The traditional set theory set intersection operator is **intersect.**

Example 5.21

"Get a list of the names of employees with the skill of chef who are assigned a duty."

```
select Name
from EMPLOYEE
where Empl_No in
   ((select Empl_No
   from EMPLOYEE
   where Skill = 'chef')
   intersect
(select Empl_No
from DUTY_ALLOCATION))
```

Name
Pierre

The result for the sample database of Figure 5.4 is given above. ■

5.3.6 Categorization

It is sometimes necessary to classify a relation into a number of groups. Each such group of tuples has a certain common property. Aggregation functions such as aver-

age, sum, and so on can be applied to each group instead of to the entire relation. SQL provides the **group by** function to allow data to be classified into categories. The aggregation functions are performed separately for each category or group. Each element in the list attached to the select clause of the select statement with the group by function must have a single value per group. The having option can be added to the group by function to specify a predicate to eliminate those elements that do not satisfy the predicate. The having option must have only one value for each group. The where clause could be used to specify predicates that would select those tuples of the relation to be considered in the categorization.

The having option usually appears with the group by function. If the having option appears without the group by function, the entire relation is treated as a single group.

Example 5.22

Consider the sample database given in Figures 5.2 and 5.4.

(a) The following SQL query generates the total charge for table 12 for the date 19860419. The result of this query for our database is the tuple (9234, 19860419, 26.00).

```
select BILL.Bill#, BILL.Day, sum(MENU.Price*ORDR.Qty)
from BILL, MENU, ORDR
where BILL.Bill# ORDR.Bill#
   and ORDR.Dish# = MENU.Dish#
   and BILL.Table# = 12
   and BILL.Day = 19860419
   group by BILL.Bill#, BILL.Day
```

We illustrate the group by function and the having option using the following queries requiring the categorization of a relation or selected tuples of the relation.

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

```
select Shift, count (distinct Empl_No)
from DUTY_ALLOCATION
group by Shift
```

Shift	count
1	4
2	3
3	1

For the DUTY_ALLOCATION relation of Figure 5.4, the result of this statement is as shown above.

(c) "Get employee numbers of all employees working on at least two dates."

```
select Empl_No
from DUTY_ALLOCATION
group by Empl_No
having count (*) > 1
```

Empl_No
123458
123461
123471

For the DUTY_ALLOCATION relation of Figure 5.4. the result of this statement is as shown above.

(d) "Get employee numbers and dates for all employees working on 19860418 and at least one other date."

```
select Empl_No, Day
from DUTY_ALLOCATION
where Empl_No in
    (select Empl_No
    from DUTY_ALLOCATION
    where Empl_No in
        (select Empl_No
        from DUTY_ALLOCATION
        where Day = 19860418)
    group by Empl_No
    having count (*) > 1)
```

Empl_no	*Day*
123458	19860418
123458	19860420
123471	19860418
123471	19860419

For the DUTY_ALLOCATION relation of Figure 5.4, the result of this statement is as shown above. Here, the inner nested select statement is used to find those employees who are working on 19860418. For our sample relation it gives the set {123457, 123458, 123471}. The where clause of the second select statement is used to eliminate tuples of DUTY_ALLOCATION where the *Empl_No* is not in the set. Only the tuples not so eliminated are considered for the grouping. The having count(*) > 1 eliminates the group of employees working only on 19860418. The result of the second select statement is the set {123458, 123471}. The outer select statement is used to provide multiple *Day* values per group ■

5.3.7 Updates

SQL includes three update statements to modify the data. These are the **insert, update,** and **delete statements.** In Section 5.3, we saw the syntax of these statements. Here we give some examples of their usage.

Example 5.23

(a) "Insert a tuple in the BILL relation with *Bill#* 9234 for *Table#* 12 on *Day* 19860419, where the waiter is 123456."

```
insert into BILL (Bill#, Day, Waiter#, Table#)
    values (9234, 19860419, 123456, 12)
```

The attributes given in the statement above are ordered differently from those in the relation scheme. The values for these attributes are given in the value clause. The remaining attributes are set to null.

(b) "Insert a DUTY_ALLOCATION tuple for *Posting_No* 321, *Empl_No* 123456, *Shift* 2, and *Day* 86/04/22."

```
insert into DUTY_ALLOCATION
    values (321, 123456, 19860422, 2)
```

The same insertion can also be specified as:

```
insert into DUTY_ALLOCATION (Empl_No, Shift, Day,
                             Posting_No)
values (123456, 2, 19860422, 321) ■
```

Note that in the second format of the insert statement, the attribute names may appear in a different order than in the relation. The latter format of the insert statement is used where data values for all the attributes are not being specified. The attributes whose values are not explicitly specified are assigned the NULL value.

Example 5.24

(a) "Copy all tuples from DUTY_ALLOCATION into NEW_DUTY_ALLOCATION," is specified as shown below. Here the attributes of NEW_DUTY_ALLOCATION are those specified in a create statement for it. In this example these attributes are compatible to those in DUTY_ALLOCATION.

```
insert into NEW_DUTY_ALLOCATION:
    select *
    from DUTY_ALLOCATION
```

(b) "Create a relation of duty records for shift 1."

```
insert into SHIFT1_DUTY_ALLOCATION:
    select *
    from DUTY_ALLOCATION
    where Shift = 1
```

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

```
update EMPLOYEE
set Pay_Rate = 1.1 * Pay_Rate
```

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

```
update EMPLOYEE
set Pay_Rate = 1.1 * Pay_Rate
where Skill = 'waiter'
```

(e) "Remove employee record for *Empl_No* 123457."

```
delete EMPLOYEE
where Empl_No = 123457
```

(f) "Remove all EMPLOYEE records and retain the relation."

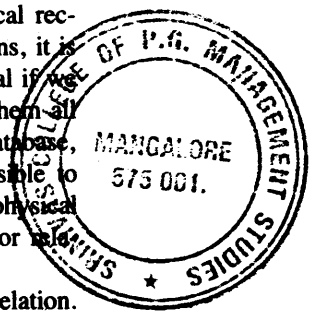
```
delete EMPLOYEE
```

(g) "Remove all EMPLOYEE records and drop the relation."

```
drop EMPLOYEE ■
```

5.4 Views: SQL

We have seen how users can manipulate the relations stored in the database. In examples presented so far, we have been manipulating the conceptual or "physical"[3] relations. Such conceptual relations are sometimes referred to as **base relations.** Corresponding to each of these base relations there exists one (or more) physical record(s) in one (or more) data file(s). Sometimes, for security and other concerns, it is undesirable to have all users see the entire relation. It would also be beneficial if we could create useful relations for different groups of users, rather than have them all manipulate the base relations. Any relation that is not a part of the physical database, i.e., a virtual relation, is made available to the users as a **view.** It is possible to create views in SQL. A relation in a view is virtual since no corresponding physical relation exists. A view represents a different perspective of a base relation or relations.



The result of a query operation on one or more base relations is a relation. Therefore, if a user needs a particular view based on the base relations, it can be defined using a query expression. To be useful, we assign the view a name and relate it to the query expression:

create view <view name> **as** <query expression>

A view is a relation (virtual rather than base) and can be used in query expressions, that is, queries can be written using the view as a relation. Views generally are not stored, since the data in the base relations may change. The base relations on which a view is based are sometimes called the **existing relations.** The definition of a view in a **create view** statement is stored in the system catalog. Having been defined, it can be used as if the view really represented a real relation. However, such a virtual relation defined by a view is recomputed whenever a query refers to it.

Example 5.25

(a) For reasons of confidentiality, not all users are permitted to see the *Pay_Rate* of an employee. For such users the DBA can create a view, for example, EMP_VIEW defined as:

create view EMP_VIEW **as**
(**select** *Empl_No, Name, Skill*
from EMPLOYEE)

(b) A view can be created for a subset of the tuples of a relation, as in this example. For assigning employees to particular jobs, the manager requires a list of the employees who have not been assigned to any jobs:

create view FREE **as**
(**select** *Empl_No*
from EMPLOYEE)

[3]By physical we mean that the relation corresponds to some stored data. This data may not be stored as a table and may actually be split horizontally or vertically and reside on one or more storage devices (at one or more sites).

Views that involve a join may or may not be updatable. Such views are not updatable if they do not include the primary keys of the base relations. When the view includes the primary keys of the base relations, the target base tuples may be identifiable and hence updatable, provided the attributes included in the views are derived using reversible operations and both the forward (from the attribute in the base relation to the view) and reverse (from the attribute in the view to the base relation) operations are known to the DBMS.

The need for allowing a view to update a relation derived from the join of two relations can be illustrated by the following example.

Consider our EMPLOYEE*(Empl_No, Name, Skill, Pay_Rate)* relation. Suppose as a result of a reorganization of the database this relation is replaced by two relations EMPL*(Empl_No, Name, Skill)* and PAYRATE*(Empl#, Pay Rate)*, defined as follows:

```
create table EMPL
    (Empl_No      integer not null,
    Name      char(25),
    Skill     char(20))
create table PAYRATE
    (Empl#      integer not null,
    Pay_Rate decimal(10,2))
```

Applications and users of the original relation EMPLOYEE continue using the database as before since they are now provided with the following view:

```
create view EMPLOYEE Empl_No, Name, Skill, Pay_Rate as
    (select Empl_No, Name, Skill, Hourly_Rate
        from EMPL, PAYRATE
            where Empl_No = Empl# )
```

The user of the EMPLOYEE relation should be insulated from this split and allowed to continue to use the database as they were accustomed to before the database reorganization. This would include making appropriate updates. If this view derived from a join could not be used to insert a tuple or make changes, then the users of the relation EMPLOYEE are not insulated from the database reorganization.

Some problems could arise when a new record is inserted in the database using a view instead of the base relation. One problem is that of assigning data values to attributes not included in the view. A method of resolving this is to insert null values for these attributes. However, this can be done only if the attributes in the base relation are defined without the not null option. If a value of a nonprimary attribute included in the view is not specified for insertion, then a null value is assigned to the corresponding attribute in the base relation. Such insertion into the base relation via the view can succeed provided the base attributes can accept a null value.

The other problem is the possibility of a record inserted by a view disappearing from that view. This is illustrated by the following example:

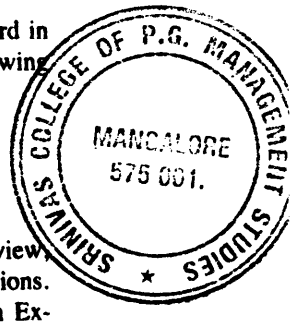
```
create view SOME_EMPLOYEE as
    (select (*)
    from EMPLOYEE
    where Empl_No < 123470)
```

The user of the view, SOME_EMPLOYEE, can insert the tuple (123481, 'Pavan', 'VP Developments', 50.00) in this relation. However, once inserted, this record will be inaccessible. Such anomalies could be avoided if the DBMS verifies that any record that is allowed to be inserted in the database satisfies the predicates of the view.

The view to be used in updates must include the primary attributes of the base relation, and these must have a nonnull value for insertion. If these conditions are not satisfied, the record to be inserted will have null values for the primary attributes. This cannot be allowed; in such cases the insertion will fail.

Any attribute in the view can be updated as long as the attribute is simple and not derived from a computation involving two or more base relation attributes. The view must, of course, include the primary attributes (or the attributes of a candidate key), otherwise the record to be updated cannot be determined and the update will fail.

The view EMP_VIEW of Example 5.25a can be used to insert a new record in the database. It is easy to see that no updates can be allowed through the following view, since it does not include the primary attribute:



create view QUALIFICATIONS **as**
(select *Name, Skill*
from EMPLOYEE)

When a view is defined on the natural join of a number of relations, the view, if used for updates, is required to include the primary keys of all base relations. Consider the view ELIGIBILITY*(Empl_No, Posting_No, Skill)*, obtained as in Example 5.8 by a join of EMPLOYEE and POSITION. It contains the primary attributes of the two relations. A tuple such as (123481, 331, cashier) inserted using this view could succeed provided no tuples with *Empl_No* = 123481 *or Posting_No* = 331 exist in the EMPLOYEE and POSITION relations. The result of the insertion would be the tuples (1234581, null, cashier, null) and (331, cashier) in the two relations.

On the other hand, consider the view ITEMIZED_BILL*(Bill#, Dish_Description, Price, Qty, Price*Qty)* created by a query such as the one given in Example 5.9 and involving the relations BILL, MENU, and ORDR. This view does not contain the primary attributes of all its underlying relations. Consider the tuple (9234, Club sandwich, 10.50, 2, 21.00) of ITEMIZED_BILL. An attempt to update *Dish_Description* will fail because the *Dish#* cannot be determined uniquely. (The club sandwich may be offered as *Dish#* 100 on the lunch menu and as *Dish#* 400 on the room service menu with different prices and both items may be included on the same bill.) An attempt to update *Price*Qty* of the club sandwich from 21.00 to 27.00 cannot be unambiguously translated into a change in the base relations. Suppose a change in *Price*Qty* is given along with a change in *Price* and *Qty* to be 27.00, 9.00, 3, respectively. It is then possible to determine, in the current state of the example database, the actual tuples to be updated by examining all the tuples of ORDR, MENU and the previous values of the tuple of ITEMIZED_BILL. Even though this update is possible in this particular example, attempts to make such an update will fail in most DBMSs. Finally, updating *Bill#* can succeed, although it is debatable if such a change should be made through a view rather than the base relation BILL.

There remains a grey area in determining if an update to a view is theoretically sound under the following conditions: the view is derived from (a) a relation that is

This allows us to declare a tuple variable and restrict it to assume values that are tuples from the relation following the keyword **is.** This relation is the domain (the set of tuple values) of the tuple variable. A reference to the tuple variable is a reference to a tuple of the relation. The use of a tuple variable is similar to that in tuple calculus wherein a tuple variable is defined by writing

<tuple variable> ϵ <relation>

The use of a tuple variable is similar to the variable declaration in programming languages where a variable is allowed to have, at a given time, a value from a set of declared values (specified by the type). The tuple variable can thus be visualized as a place marker in our relation.

Example 5.27

range of d **is** DUTY_ALLOCATION
range of e **is** EMPLOYEE

The tuple variables d and e, at any given time, refer to a tuple in the DUTY ALLOCATION and EMPLOYEE relations, respectively. ■

In Chapter 4, we used RELATION_NAME[*Attribute_Name*] to refer to the values of an attribute of a relation. In QUEL this requires the use of qualified names:

RELATION_NAME.*Attribute_Name*, or
Tuple_Variable.*Attribute_Name*

The period is used to qualify the attribute by the relation. Note that in the convention followed in Chapter 4, a group of attribute names could be specified within brackets. There is no such simple grouping technique in QUEL.

Example 5.28

We assume that the tuple variable d has been declared as in Example 5.27. Then,

d.*Posting_No*

refers to the value of the *Posting_No* attrbitue of a tuple in the DUTY_ ALLOCATION relation. ■

Index Statement

The indexes are defined for an existing relation using the **index** statement. It specifies the name of the secondary index to be built and the attributes from the relation that are used for indexing. The purpose of creating a secondary index is to increase the efficiency of secondary key retrieval. A relation could have any number of secondary indexes created for it in addition to the index created on the primary key. All indexes are destroyed when the relation is destroyed. Once created, an index is maintained and used automatically by the DBMS. The syntax of the index statement is as follows:

index on <relation name> **is** index_name
(attribute_name [,attribute_name, . . .])

Example 5.29 The following statement creates an additional index named *nameindex* for the EMPLOYEE relation using the *Name* attribute:

index on EMPLOYEE **is** *nameindex (Name)* ■

Destroy Statement

The **destroy** statement is used in QUEL to eliminate a relation, index, or view (discussed in Section 5.7.9). The syntax of the destroy statement is:

destroy <name[,name, . . .]>

where each name is the name of an existing relation, index, or view.

Example 5.30 The following statement destroys the index named *nameindex:*

destroy *nameindex* ■

Modify Statement

The **modify** statement is used to modify the storage structure of a relation from the current one to that specified in the statement. The storage structures supported in INGRES are B-tree, hash, ISAM, and heap. The compressed versions of these storage structures are also supported; the compression is on the physical storage medium. One example of a compression scheme is to suppress the trailing blanks of a character string. The syntax of the modify statement is as follows:

modify relation_name to storage_structure [**on** attribute1 [**order ascending|descending**] [, . . . ,]]

Here the name of the relation is specified by relation_name and the new storage structure by storage_structure. The **on** clause indicates the attribute(s) to be used for ordering the relation. The order can be specified optionally as ascending or descending; ascending being the default. If the on clause is not specified, ascending order of the relation by the first attribute is assumed.

Example 5.31 The following statement modifies the storage structure of the EMPLOYEE relation to a compressed hash (chash) structure with *Empl_No* as the hash key:

modify EMPLOYEE to chash on *Empl_No* ■

Figure E Modified form of Figure C.

SALARY:

Empl_No	*Pay_Rate*	*Hours*
123456	7.50	40.5
123457	8.79	42.5
123458	4.70	47.5
123459	4.90	0.0
123460	4.70	48.0
123461	9.00	48.0
123471	14.00	42.7
123472	14.50	45.5

(i)

result:

Empl_No	*Gross_Pay*
123456	303.50
123457	373.58
123458	223.25
123460	225.60
123461	432.00
123471	597.80
123472	659.75

(ii)

in part ii of Figure E (the second column heading has been renamed *Gross_Pay* instead of *Pay_Rate*Hours*):

range of s **is** SALARY
retrieve (s.*Empl_No*, *Gross_Pay* = s.*Pay_Rate*s.Hours*)
where s.*Hours* > 0.0 ■

5.7.4 Multiple Variable Queries

So far we have expressed queries using a single tuple variable and these queries required information from a single relation. However, when we are required to retrieve information stored in multiple relations we need to use multiple variables—one tuple variable for each relation. In this section we give examples of queries that require the use of multiple variables.

Example 5.37 "Get the name of the waiter for table 17, identified as *Waiter_Name*."

range of e **is** EMPLOYEE
range of b **is** BILL
retrieve (*Waiter_Name* = e.*Name*)
where e.*Empl_No* = b.*Waiter#* **and** b.*Table#* = 17 ■

In this query we get the identifier for waiter assigned to table 17 and compare it with the employee identifier of employee tuples (the attribute *Waiter#* in BILL refers to the same instance of the entity set employee as attribute *Empl_No* in EMPLOYEE). For the relations MENU and EMPLOYEE of Figures 5.2 and 5.4, the result of this query is the name Ian.

Example 5.38

"Get shift details of the employee named Pierre."

```
range of d is DUTY_ALLOCATION
range of e is EMPLOYEE
retrieve (d.Posting_No,d.Shift,d.Day )
where d.Empl_No = e.Empl_No and e.Emp_Name = 'Pierre'
```

■

The use of multiple variables is not restricted to different relations. Sometimes it becomes necessary to declare multiple tuple variables over the same relation. Thus if we want to compare the tuples of the same relation, we can have several tuple variables ranging over the relation. We demonstrate this in the following example.

Example 5.39

"Find employees whose rate of pay is more than that of employee Jon."

In this query, at any given time, we need data on two employees: one is fixed (the data for employee Jon) and the other will be another employee. Thus, we need one tuple variable that can be used to refer to the tuple for employee Jon, and another tuple variable for the other employee. (Imagine that this second tuple variable will be used to scan the complete relation, one tuple at a time.)

```
range of e is EMPLOYEE
range of e1 is EMPLOYEE
retrieve (e.Name,e.Pay_Rate)
where e.Pay_Rate > e1.Pay_Rate
   and e1.Name = 'Jon'
```

Name	*Pay_Rate*
Ian	9.00
Pierre	14.00
Julie	14.50

The tuple variable e_1 has the data for employee Jon while at any given instance the tuple variable e has data for another employee. The result of this query is shown in the example. ■

Example 5.40

"Get all pairs of *Empl_No* with the **same** *Posting_No*."

```
range of d is DUTY_ALLOCATION
range of d1 is DUTY_ALLOCATION
retrieve (d.Empl_No, d1.Empl_No )
where d.Posting_No = d1.Posting_No
      and (d.Empl_No < d1.Empl_No )
```

Empl_No	*Empl_No*
123456	123461
123456	123461

■

In this query we need to compare two tuples of the DUTY_ALLOCATION relation. The condition (d.*Empl_No* < d_1.*Empl_No*) guarantees that only unique employee pairs are retrieved. Employee 123458, who is posted twice to *Posting_No* 323, is not in the result since the *Empl_Nos* are the same. Also, by using this condition we avoid including symmetrical tuples in the result. Thus the tuple (123461, 123456) is excluded from the result. (In Example 5.39 we did not need to specify such a condition). Note, however, that the result shown above does have duplicate tuples because *Posting_No* 321 is associated with *Empl_No* 123456 twice in the

relation DUTY_ALLOCATION. We could use the unique option in the retrieve statement to remove such duplicate tuples.

We next illustrate a query requiring the join of three relations:

Example 5.41

Consider the requirement to generate the itemized bill for table 12 for the date 19860419. This requires details from three relations, BILL, ORDR, and MENU. The itemized bill can be generated using the statements given below. The result of the query on the relations given in Figure 5.2 is also shown.

```
range of b is BILL
range of m is MENU
range of o is ORDR
retrieve (b.Bill#,m.Dish_Description,m.Price, o.Qty,
                    Dish_Total = m.Price*o.Qty)
where b.Bill# = o.Bill#
   and o.Dish# = m.Dish#
   and b.Table# = 12
   and b.Day = 19860419
```

Bill#	*Dish_Description*	*Price*	*Qty*	*Dish_Total*
9234	Coffee	2.50	2	5.00
9234	Club sandwich	10.50	2	21.00

■

QUEL does not allow nested retrieve statements (similar to the nested select statement) and hence unlike SQL this method cannot be used to generate the itemized bill.

5.7.5 Set Operations in QUEL

The set operations, for example union and intersection, are not supported by QUEL. A number of queries require us to use some of these operators. In relational calculus a tuple variable can be declared independent of the relation and thus can accept values from different relations. In QUEL a qualified tuple variable appears in the target list and since the tuple variable ranges over a single relation, we need some explicit mechanism for creating unions. The same holds true for the other operations. In Section 5.7.8 we introduce some of the data modification commands, and show how they can be used to encode the set operations indirectly.

5.7.6 Aggregation Operators in QUEL

QUEL provides a number of aggregation operators to be used in expressions. These allow a user to perform computations on the values of the relation's attributes.