Chapter 15

RELATIONAL QUERY LANGUAGES

In this chapter we give a brief overview of several query languages from various relational database systems. We shall not give a complete exposition of the languages. Our point is, rather, to give the flavor of each, show how they are based on the algebra, calculus, or tableaux, and indicate where they depart from the relational model as we have defined it.

We shall look at five languages: ISBL, from the PRTV system; QUEL, from INGRES; SQL, from System R; QBE, which runs atop several database systems; and PIQUE, from the experimental PITS system. ISBL is based on relational algebra, QUEL and SQL resemble tuple calculus, and QBE is a domain calculus-like language, with a syntax similar to tableau queries. PIQUE is a tuple calculus-like language, but it presents a universal relation scheme interface through the use of window functions.

For practical reasons and usability considerations, relational query languages do not conform precisely to the relational model. They all contain features that are extensions to the model, and some have restrictions not present in the model. Nearly all relational systems have facilities for virtual relation definition. Languages based on tuple and domain calculus must allow only safe expressions. Safety is usually guaranteed by the absence of explicit quantifiers or by having variables range over relations rather than all tuples on a given scheme. Query languages often allow a limited amount of arithmetic and string computation on domain values, and sometimes handle sets of values through aggregate operators (count, average, maximum) and set comparisons. As mentioned in the last chapter, QBE contains operators for dealing with transitive closures. In the theory, order of attributes and tuples within a relation is immaterial. Query languages give control over attribute order and sort order for tuples when listing relations. In addition to output commands, query languages usually contain some form of assignment statement to store intermediate results. Some languages also give explicit control over duplicate tuple elimination after projection, so the data type they support is actually multisets of tuples. Suppression and invocation of duplicate removal are useful in connection with aggregate operators. Relational languages sometimes have special constructs or alternative syntax for use from within a regular programming language.

All the query languages we examine are as powerful as relational algebra. In some cases, however, this power may only be achieved through use of a sequence of assignment statements, and not within a single expression in the language. Sometimes, data manipulation commands must be used to get the effect of certain algebraic operators.

We shall not be exhaustive in demonstrating the features of the languages we present here. Some of the languages have facilities for data manipulation in addition to querying; we shall consider only the "query" portion of those languages. We shall not spend much time on formal syntax or semantics of the languages studied, but rather work by example. To make comparisons between the languages easy, all examples in this chapter will be based on the database in Figure 15.1, which is the same database as was used for most examples in Chapter 10.

15.1 ISBL

ISBL (Information System Base Language) is an algebra-based query language used in the PRTV (Peterlee Relational Test Vehicle) system. PRTV is an experimental, interactive database system developed at IBM's United Kingdom Scientific Centre.

ISBL expressions are built up with six operators, which correspond to algebraic operators or generalizations of them. Union, intersection, and join are denoted as +, ., and *, respectively, and behave exactly as in the algebra. Difference, denoted by −, has been generalized to be an "antijoin" operator. For relations r and s, with possibly different schemes,

$$r - s = r - (r \ltimes s),$$

where the − on the left is the one from ISBL and the one on the right is the usual one from relational algebra. Thus, in ISBL, $r - s$ is all the tuples in r that join with no tuple in s. The selection $\sigma_C(r)$ is written as $r\!:\!C$ in ISBL, where C is a selection condition built up from attribute names and constants with the comparators in $\{=, \neq, >, \geq, \leq, <\}$, parentheses, and the logical connectives & (and), | (or), and $\neg$ (not). Projection and renaming are combined into one operation in ISBL. The projection $\pi_{ABC}(r)$ is r % A, B, C in ISBL. To rename an attribute A_1 to be A_2 during projection, A_1 −> A_2 is used in place of A_1 to the right of %. A rename by itself is accomplished using % with all the attributes in the relation on the right. Three dots can be

pinfo(PART#	SUBPARTOF	PARTNAME)
211	0	coach seat
2114	211	seat cover
2116	211	seat belt
21163	2116	seat belt buckle
21164	2116	seat belt anchor
318	21164	funny little bolt
206	0	overhead console
2061	206	paging switch
2066	206	light switch
2068	206	air nozzle

usedon(PART#	PTYPE	NUSED)
211	707	86
211	727	134
2114	707	86
2114	727	134
2116	707	244
2116	727	296
21164	707	488
21164	727	592

instock(PART#	LOCATION	QUANTITY)
211	JFK	106
211	Boston	28
211	O'Hare	77
2114	JFK	6
2114	O'Hare	28
2116	Boston	341
2116	O'Hare	29
21164	Atlanta	36,391

Figure 15.1

used to stand for all unnamed attributes in a relation. Thus, if we want to rename just B in r, we would write r % B $->$ D, ... Expressions in ISBL are left-associative; parentheses may be used to override that associativity.

The keyword **list** is used to print the result of an expression in ISBL, and the result of an expression can be assigned to a relation using $=$.

To keep examples in this chapter concise, many of them will contain just an English statement of a query and the corresponding query language statement, with the result of evaluating the query included sometimes.

Example 15.1 What are the names of subparts of part 2116?

```
list pinfo : PARTNAME# = 2116 % PARTNAME
```

```
      (PARTNAME   )
      ------------
      seat belt buckle
      seat belt anchor
```

Example 15.2 How many seat belts are at Boston?

```
list pinfo * instock : PARTNAME = "seat belt" &
    LOCATION = "Boston"% QUANTITY
```

```
      (QUANTITY)
      ----------
         341
```

Example 15.3 Which parts are not in stock at either Boston or O'Hare?

```
list pinfo − (instock : LOCATION = "Boston"|
    LOCATION = "O'Hare")% PART#
```

```
      (PART#)
      -------
       21163
       21164
         318
         206
        2061
        2066
        2068
```

Example 15.4 What are all the subparts of subparts of seat belts?

```
list pinfo % PART#−>SUBSUBPART, SUBPARTOF−>PART#*
    pinfo : PARTNAME = "seat belt" % SUBSUBPART
```

```
      (SUBSUBPART)
      ------------
         21163
         21164
```

Example 15.5 Which parts used on a 707 are in stock at Atlanta?

> **list** *usedon* : *PTYPE* = "707" % *PART#*
> (*instock* : *LOCATION* = "Atlanta" % *PART#*)

(PART#)
21164

We could also perform the same query by assigning intermediate results to relations.

> *r*1 = *usedon* : *PTYPE* = "707" % *PART#*
> *r*2 = *instock* : *LOCATION* = "Atlanta" % *PART#*
> *r*3 = *r*1 . *r*2
> **list** *r*3

ISBL can support virtual relations through use of a delayed evaluation feature. No expression is evaluated in ISBL until its result is needed. Relation names in expressions are normally bound to the current value of the relation. Prefixing a relation name with N! indicates that the value of the relation to be used is the current value at the time of evaluation.

Example 15.6 Suppose we want a virtual relation *stock*707 that gives part number, number used, location, and quantity for all parts used in a 707. If we make the assignment

> *stock*707 = N!*usedon* * N!*instock* : *PTYPE* = "707" %
> *PART#, NUSED, LOCATION, QUANTITY*

then any time we write **list** *stock*707 or *stock*707 occurs in an expression being evaluated, the current values of *usedon* and *instock* are used. If the N! is removed in both places, *stock*707's value will depend on the state of *usedon* and *instock* at the point when *stock*707 is defined.

ISBL does not have any operators within itself to perform computation on values. However, computed relations, whose extensions are functions or procedures in a general-purpose programming language, can be used to provide computation. The extension of a computed relation in ISBL takes one of two forms. One is a Boolean-valued function that recognizes those tuples that are in the relation, which is a slight departure from what we saw in Section 14.4.

The other form is a procedure that takes some values in a tuple as inputs and returns others as output.

Example 15.7 Let *adequate*(a, b) be a function that returns *true* exactly when $a \geq .5b$. The ISBL statement

> **list** *usedon* * *instock* * *adequate*(*QUANTITY, NUSED*) %
> *PART#, PTYPE, LOCATION*

gives the locations that have at least half the number of a part used on a given aircraft:

(PART#	PTYPE	LOCATION)
211	707	JFK
211	727	JFK
211	707	O'Hare
211	727	O'Hare
2116	707	Boston
2116	727	Boston
21164	707	Atlanta
21164	727	Atlanta

Example 15.8 Let *needed*(a, b, c) be a procedure that takes a and b as input and returns $c = max(.5b - a, 0)$. The statement

> **list** *usedon* * *instock* * *needed*(*QUANTITY, NUSED* |
> *AMT*:*ORDER*) : *PTYPE* = "727" & *ORDER* > 0 % *PART#,*
> *LOCATION, ORDER*

gives the number of each part to order at each location to bring the supply up to half the number used on a 727 (provided that the location already stocks the part):

(PART#	LOCATION	ORDER)
211	Boston	39
2114	JFK	61
2114	O'Hare	39
2116	O'Hare	119

In the call to *needed*, the vertical bar separates input parameters from output parameters, and *AMT*: *ORDER* indicates that the output parameter is supposed to be a new attribute *ORDER* with domain *AMT*.

ISBL also contains features for passing entire relations to and from procedures for updating, computing aggregates, and formatting output.

15.2 QUEL

QUEL (QUEry Language) is the data manipulation language for the INGRES (INteractive Graphics and REtrieval System) database system. INGRES is a fairly complete relational system that was developed at the University of California at Berkeley, and it is still being revised and extended. QUEL, in addition to retrieval functions, contains commands for update, authorization, integrity, and view definition. We shall only cover the retrieval aspects of QUEL.

QUEL is based on the tuple relational calculus. Tuple variables are all existentially quantified and bound to relations. Tuple variables are declared and bound with a statement of the form,

range of ⟨tuple variable⟩ **is** ⟨relation name⟩.

The A-component of a tuple variable x is denoted as $x.A$.

Example 15.9 For subsequent examples in this section, we shall assume the following bindings:

range of p **is** *pinfo*
range of $p1$ **is** *pinfo*
range of u **is** *usedon*
range of i **is** *instock*

With these bindings, *p.PART#*, *u.NUSED* and *i.PART#* are all proper references to tuple variable components.

The basic form of a retrieval in QUEL is

retrieve (⟨target list⟩) **where** ⟨condition⟩.

The condition corresponds to the formula in a tuple calculus expression. The target list is a sequence of tuple variable components, which resembles the portion of of a domain-calculus expression to the left of the bar.

Example 15.10 What is the name of part 2116?

> **retrieve** (*p.PARTNAME*) **where** *p.PART#* = 2116

(PARTNAME)
seat belt

Example 15.11 What are the names and quantities of parts at O'Hare?

> **retrieve** (*p.PARTNAME, i.QUANTITY*)
> **where** *p.PART#* = *i.PART#* and *i.LOCATION* = "O'Hare"

(PARTNAME	QUANTITY)
coach seat	77
seat cover	28
seat belt	29

The comparators =, !=, >, >=, <= and < are allowed in comparisons, and the logical connectives **and**, **or**, and **not** can be used to combine comparisons. The attributes for the result of a retrieval are taken from the corresponding tuple variable components. The attributes in the result relation can be changed by using *NEWNAME* = *x.OLDNAME* in the target list. Renaming must be done if the resulting relation would have the same attribute twice.

Example 15.12 Which parts are subparts of the same part?

> **retrieve** (*SUBPART1* = *p.PART#*, *SUBPART2* = *p1.PART#*)
> **where** *p.SUBPARTOF* = *p1.SUBPARTOF* **and**
> *p.SUBPARTOF* != 0 **and** *p.PART#* < *p1.PART#*

(SUBPART1	SUBPART2)
2114	2116
21163	21164
2061	2066
2061	2068
2066	2068

The condition *p.PART#* < *p1.PART#* is included to prevent an extra tuple for each pair of parts and to eliminate pairing a part with itself.

The condition part of a query is optional. The keyword **all** can be used to represent all components of a tuple variable in the target list. Thus,

retrieve *u*.**all**

returns the entire *usedon* relation. Computational expressions can appear most places that a tuple variable component would be appropriate. If an expression appears in the target list, it must be renamed.

Example 15.13 Which locations have at least half the number of a part used in a given aircraft?

```
retrieve (u.PART#, u.PTYPE, i.LOCATION)
    where u.PART# = i.PART# and u.NUSED * 0.5
        <= i.QUANTITY
```

Example 15.14 What proportion of the number of coach seats used on a 707 does each location have?

```
retrieve (i.LOCATION, PROPORTION =
    (i.QUANTITY/u.NUSED))
    where p.PART# = u.PART# and p.PARTNAME
        = "coach seat" and u.PART# = i.PART# and
        u.PTYPE = "707"
```

(LOCATION	PROPORTION)
JFK	1.233
Boston	.326
O'Hare	.894

Assignment of the result of a retrieval to a relation is achieved via the notation **into** ⟨relation⟩ after **retrieve.**

Example 15.15

retrieve into *used*727 (*u*.**all**) **where** *u*.*PTYPE* = "727"

makes *used*727 a relation with the same scheme as *usedon*, but consisting only of 727 information.

QUEL provides the aggregation operators **count**, **min**, **max**, **avg**, and **sum**, which can be used in expressions.

Example 15.16 How many different parts are there?

retrieve (*NUMPARTS* = **count**(*p.PART#*))

(NUMPARTS)
10

Example 15.17 Which part has the most in stock and which part has the least in stock at any one location?

retrieve (*i.PART#, i.LOCATION*)
  where *i.QUANTITY* = **max**(*i.QUANTITY*) **or** *i.QUANTITY* =
    min(*i.QUANTITY*)

(PART#	LOCATION)
2114	JFK
21164	Atlanta

The aggregate operators **count**, **avg**, and **sum** have "unique" versions, distinguished by a "**u**" on the end, that eliminate duplicates before applying the operator.

Example 15.18 How many locations have parts?

retrieve (*NUMLOCS* = **countu**(*i.LOCATION*))

(NUMLOCS)
4

Using **count** in place of **countu** in this query would produce the answer 8.

The component to which an aggregate operator is applied can be qualified. However, the qualification is local, and is not affected by the rest of the query.

Example 15.19 How many of part 211 are in stock?

retrieve (*TOTAL* = **sum**(*i.QUANTITY* **where** *i.PART#* = 211))

(TOTAL)
211

Example 15.20 The following query does *not* answer the question: How many seat belts are in stock?

retrieve (*NUMBELTS* = **sum**(*i.QUANTITY*))
where *p.PART#* = *i.PART#* **and** *p.PARTNAME* = "seat belt"

The **sum** is computed independently of the rest of the query.

QUEL has a grouping feature for aggregates, invoked with the keyword **by** within the argument to an aggregate operator. The component following the **by** is linked to the rest of the query.

Example 15.21 How many of each part are there?

retrieve (*i.PART#*, *TOTAL* = **sum**(*i.QUANTITY* **by** *i.PART#*))

(PART#	TOTAL)
211	211
2114	34
2116	370
21164	36,391

Example 15.22 How many seat belts are there in stock?

retrieve (*NUMBELTS* = **sum**(*i.QUANTITY* **by** *i.PART#*))
where *p.PART#* = *i.PART#* **and** *p.PARTNAME* = "seat belt"

(NUMBELTS)
370

Example 15.23 How many of each part over the maximum needed by any aircraft does each location have in stock?

retrieve (*u.PART#*, *i.LOCATION*, *OVERSTOCK* =
  (*i.QUANTITY* − **max**(*u.NUSED* **by** *u.PART#*))
 where *u.PART#* = *i.PART#* **and**
  (*i.QUANTITY* − **max**(*u.NUSED* **by** *u.PART#*)) > 0

(PART#	LOCATION	OVERSTOCK)
2116	Boston	45
21164	Atlanta	35,799

15.3 SQL

SQL (Structured Query Language) is the data manipulation language for the System R database system. System R is a prototype relational database system developed at the IBM San Jose Research Laboratory. A commercial IBM product, SQL/Data System, is based on the System R prototype. While SQL is a complete data manipulation language, we cover only its retrieval capabilities.

SQL's syntax resembles tuple calculus, though not so closely as that of QUEL. SQL's precursor is SQUARE, which resembles relational algebra in some aspects and tuple calculus in others. The main operator in SQUARE is the *mapping*, which is a selection followed by a projection. The mapping is carried over into the basic syntax of SQL, which is

select ⟨attribute list⟩
from ⟨relation⟩
where ⟨condition⟩

Example 15.24 What are the names and numbers of subparts of part 211?

select *PARTNAME*, *PART#*
from *pinfo*
where *SUBPARTOF* = 211

(PARTNAME	PART#)
seat cover	2114
seat belt	2116

Example 15.25 What parts are at Boston or O'Hare?

> **select** *PART#*
> **from** *instock*
> **where** *LOCATION* = "Boston" **or** *LOCATION* = "O'Hare"

(PART#)
211
211
2114
2116
2116

The logical connectives **and**, **or**, and **not** can be used to combine comparisons. As we see from the last example, SQL does not automatically eliminate duplicates. Duplicates can be removed by including the keyword **unique** after **select**.

Example 15.26 To get rid of duplicate entries in the last example, we can use the query

> **select unique** *PART#*
> **from** *instock*
> **where** *LOCATION* = "Boston" **or** *LOCATION* = "O'Hare"

(PART#)
211
2114
2116

SQL has constructs for tests involving sets of values and sets of tuples. A set can be listed explicitly, or it can be the result of a subquery. Tests can be made for membership, emptiness, inclusion, and comparison with members of a set, one at a time.

Example 15.27 What parts are at Boston or O'Hare?

> **select unique** *PART#*
> **from** *instock*
> **where** *LOCATION* **in** ("Boston", "O'Hare")

Example 15.28 What are the names of parts at JFK?

```
select PARTNAME
from pinfo
where PART# in
     (select PART#
      from instock
      where LOCATION = "JFK")
```

```
(PARTNAME)
----------
coach seat
seat cover
```

Example 15.29 Which location has the fewest of part 211?

```
select LOCATION
from instock
where QUANTITY <=all
     (select QUANTITY
      from instock
      where PART# = 211)
```

```
(LOCATION)
----------
  Boston
```

Example 15.30 Which locations stock all the parts that Boston does?

```
select LOCATION
from instock
where set(PART#) contains
     (select PART#
      from instock
      where LOCATION = "Boston")
```

```
(LOCATION)
----------
  Boston
  O'Hare
```

The notation **set**(*PART#*) refers to the set of all *PART#*-values occurring with a *LOCATION*-value. It is possible for a subquery to reference fields from the relation in the containing query. If there are identical field names in the containing query and subquery, they are qualified with the relation name. In essence, the relation name serves as a tuple variable bound to the relation.

Example 15.31 Which locations have at least as many of a part as are used on a 707?

```
select PART#, LOCATION
from instock
where PART# in
      (select PART#
       from usedon
       where PTYPE = "707" and NUSED <= QUANTITY)
```

Note that *QUANTITY* refers to *instock.*

PART#	LOCATION
211	JFK
2116	Boston
21164	Atlanta

Another formulation of this question is

```
select PART#, LOCATION
from instock
where QUANTITY >= any
      (select NUSED
       from usedon
       where PTYPE = "707" and instock.PART# = usedon.PART#)
usedon.PART#)
```

When two copies of a relation are used in a query, the fields of each cannot be distinguished by prefixing the relation name. In these cases, an alternate qualifier may be listed after the relation name in the **from**-clause.

Example 15.32 What parts are available at more than one location?

```
select unique PART#
from instock i
where PART# in
     (select PART#
      from instock
      where LOCATION ¬= i.LOCATION)
```

```
(PART#)
  211
 2114
 2116
```

In previous examples, we have been specifying joins by the use of **in**. This method for taking joins will not work if the answer to the query involves fields from more than one relation. In that case, multiple relations are used in the **from**-clause.

Example 15.33 What are the names and quantities of parts at JFK?

```
select PARTNAME, QUANTITY
from pinfo, instock
where pinfo. PART# = instock.PART# and LOCATION = "JFK"
```

```
(PARTNAME   QUANTITY)
 coach seat      106
 seat cover        6
```

Example 15.34 What parts are subparts of the same part?

```
select p1.PART#, p2.PART#
from pinfo p1, pinfo p2
where p1.SUBPARTOF = p2.SUBPARTOF and
     p1.SUBPARTOF ¬= 0 and p1.PART# < p2.PART#
```

(*p*1.PART#	*p*2.PART#)
2114	2116
21163	21164
2061	2066
2061	2068
2066	2068

SQL allows queries to be combined with the set operators **union**, **intersect**, and **minus**. Duplicates are removed after computing **union**.

Example 15.35 Which parts are not stocked by any location?

> **select** *PART#*
> **from** *pinfo*
> **minus**
> **select** *PART#*
> **from** *instock*

(PART#)
21163
318
206
2061
2066
2068

The order of columns in the result of a query is taken from the order of the attributes in the **select**-clause. The order of the tuples can be controlled with an **order by**-clause, which contains a list of attributes from the result of the query, each followed by **asc** or **desc**, for ascending or descending order. Another feature of SQL, demonstrated in the next example, is the use of * to stand for all the attributes in a relation.

Example 15.36 The following query lists the *instock* relation by decreasing quantity and increasing part number.

> **select** *
> **from** *instock*
> **order by** *QUANTITY* **desc**, *PART#* **asc**

(PART#	LOCATION	QUANTITY)
21164	Atlanta	36,391
2116	Boston	341
211	JFK	106
211	O'Hare	77
2116	O'Hare	29
211	Boston	28
2114	O'Hare	28
2114	JFK	6

SQL allows arithmetic expressions and provides the aggregate operators **avg**, **min**, **max**, **sum**, and **count**.

Example 15.37 Which locations have at least half the number of a part used in a given aircraft?

```
select usedon.PART#, PTYPE, LOCATION
from usedon, instock
where usedon.PART# = instock.PART# and
    NUSED * 0.5 <= QUANTITY
```

Example 15.38 How many coach seats are in stock?

```
select sum (QUANTITY)
from instock
where PART# in
    (select PART#
     from pinfo
     where PARTNAME = "coach seat")
```

( )
211

SQL does not name columns corresponding to aggregates or expressions. If an **order by**-clause must refer to such a column, it uses an integer denoting the position of the column.

SQL uses a **group by**-clause to partition the tuples in a result before application of an aggregate operator. A **having** clause may be included to remove some groups of tuples.

Example 15.39 How many of each part are there?

> **select** *PART#*, **sum**(*QUANTITY*)
> **from** *instock*
> **group by** *PART#*

Example 15.40 For which parts is the total in stock at least the number used by a 707, and by what amount does the total exceed the number used?

> **select** *usedon.PART#*, *NUSED* − **sum**(*QUANTITY*)
> **from** *usedon*, *instock*
> **where** *usedon.PART#* = *instock.PART#* **and** *PTYPE* = "707"
> **group by** *usedon.PART#*
> **having** *NUSED* − **sum**(*QUANTITY*) >= 0

(PART#	)
211	125
2116	136
21164	35,903

15.4 QBE

QBE (Query-By-Example) is a relational data manipulation language designed by M. M. Zloof at IBM's Watson Research Center. A subset of the language has been implemented to run with various IBM systems. We cover the retrieval aspects of QBE. QBE also has update operations, authorization and integrity mechanisms, domain declarations, and view definition facilities.

The syntax of QBE is two-dimensional. Queries are formed by filling in a *skeleton*, which contains a relation name and its attributes, such as

pinfo	PART#	SUBPARTOF	PARTNAME

The skeleton is filled in with rows of constants and variables. A filled-in skeleton has a syntax and semantics reminiscent of tableau queries. Tableau queries, in turn, can be readily described in domain calculus. A row in a QBE query, such as

pinfo	PART#	SUBPARTOF	PARTNAME
	a	*b*	*c*

corresponds to the atom *pinfo*(*a b c*) in domain calculus.

Variables in QBE are existentially quantified, and are represented by underlined strings. The particular name given a variable in no way affects the interpretation of a query, although it is usual to use example values from the domain of an attribute as names. Strings without underlines are constants. The operator **P.**, for *print*, is prefixed to any variable or constant to appear in the result of the query. **P.** is essentially a mechanism to form the equivalent of a summary in a tableau query, without having to write a separate row.

Example 15.41 Which locations stock part 211?

instock	PART#	LOCATION	QUANTITY
	211	**P.** $\underline{\text{Chicago}}$	$\underline{25}$

Result:

instock	LOCATION
	JFK
	Boston
	O'Hare

Note that the name $\underline{\text{Chicago}}$ for a variable has no effect on the values ultimately retrieved.

If a variable is mentioned in only one place, it may be omitted. QBE assumes each blank slot in a row contains a unique variable.

Example 15.42 The query in the last example could be written

instock	PART#	LOCATION	QUANTITY
	211	**P.**	

Selections with comparators other than equality are done by prefixing a constant or variable with the comparator. Essentially, θa, for comparator θ, represents the subset of the domain of the column equal to $\{c \mid c\ \theta\ a\}$.

Example 15.43 Which parts have more than 50 in stock at some location?

instock	PART#	LOCATION	QUANTITY
	P.		>=50

Result:

instock	PART#
	211
	2116
	21164

Note that QBE does eliminate duplicates.

Example 15.44 For which parts are more than 100 used on an aircraft other than a 727?

usedon	PART#	PTYPE	NUSED
	P.	¬= 727	>= 100

Result:

usedon	PART#
	2116
	21164

Queries are not limited to one row. Multiple rows may be used.

Example 15.45 Which parts are in stock at Boston and O'Hare?

instock	PART#	LOCATION	QUANTITY
	P. $\underline{100}$	Boston	
	$\underline{100}$	O'Hare	

Result:

instock	PART#
	211
	2116

Using the print operator in multiple rows gives the union of the results specified by each row.

Example 15.46 What parts are in stock at Boston or O'Hare?

instock	PART#	LOCATION	QUANTITY
	P. $\underline{100}$	Boston	
	P. $\underline{101}$	O'Hare	

Result:

instock	PART#
	211
	2114
	2116

Example 15.47 The following query retrieves information on part 211.

pinfo	PART#	SUBPARTOF	PARTNAME
	P. 211	P.	P.
	P.	P. 211	P.

Result:

pinfo	PART#	SUBPARTOF	PARTNAME
	211	0	coach seat
	2114	211	seat cover
	2116	211	seat belt

The print operator can be applied to an entire row by placing it at the left end of the row.

Example 15.48 The query in the last example can be written as follows.

pinfo	PART#	SUBPARTOF	PARTNAME
P.	211		
P.		211	

As part of a condition, it is possible to specify that no tuple matching a certain row may appear in a relation. No portion of such a row may be printed, however. The ability to test for the absence of a tuple is not available with tableau queries.

Example 15.49 Which locations have parts that Boston does not, and what are the parts?

instock	PART#	LOCATION	QUANTITY
	P. $\underline{100}$	**P.** $\underline{\text{Chicago}}$	
	$\underline{100}$	Boston	

Result:

instock	PART#	LOCATION
	2114	JFK
	2114	O'Hare
	21164	Atlanta

QBE also has the facility for matching substrings of string values by concatenating variables and constants.

Example 15.50 The query

pinfo	PART#	SUBPARTOF	PARTNAME
	P.		**P.** seat$\underline{\text{belt}}$

finds all parts where the partname begins with "seat."
Result:

pinfo	PART#	PARTNAME
	2114	seat cover
	2116	seat belt
	21163	seat belt buckle
	21164	seat belt anchor

With tableau queries, rows are bound to various relations with tags. In QBE, a separate skeleton is used for each relation involved in a query.

Example 15.51 What are the names of parts at JFK?

pinfo	PART#	SUBPARTOF	PARTNAME
	$\underline{100}$		**P.** $\underline{\text{bolt}}$

instock	PART#	LOCATION	QUANTITY
	$\underline{100}$	JFK	

Example 15.52 Which locations have at least as many of a part as are used on a 707?

usedon	PART#	PTYPE	NUSED
	$\underline{100}$	707	$\underline{50}$

instock	PART#	LOCATION	QUANTITY
	P. $\underline{100}$	**P.** $\underline{\text{Chicago}}$	$>= \underline{50}$

If values from multiple relations must be combined, variables in the same column of one relation are to appear in different columns, or a column must be renamed, it is necessary to specify an additional relation for the result.

Example 15.53 What are the names and quantities of parts at JFK?

pinfo	PART#	SUBPARTOF	PARTNAME
	$\underline{100}$		$\underline{\text{bolt}}$

instock	PART#	LOCATION	QUANTITY
	$\underline{100}$	JFK	$\underline{50}$

jfkparts	PARTNAME	QUANTITY
P.	$\underline{\text{bolt}}$	$\underline{50}$

Result:

jfkparts	PARTNAME	QUANTITY
	coach seat	106
	seat cover	6

Sometimes it is impossible or inconvenient to specify all the constraints among variables with a skeleton. QBE provides an auxiliary condition box to hold additional constraints.

Example 15.54 Which parts are subparts of the same part?

pinfo	PART#	SUBPARTOF	PARTNAME
	$\underline{100}$	$\underline{200}$	
	$\underline{101}$	$\underline{200}$	

CONDITIONS
$\underline{200} \neg = 0$
$\underline{100} < \underline{101}$

subparts	SUBPART1	SUBPART2
P.	$\underline{100}$	$\underline{101}$

Result:

subparts	SUBPART1	SUBPART2
	2114	2116
	21163	21164
	2061	2066
	2061	2068
	2066	2068

The order of tuples in a result can be controlled by the prefixes **AO.** (ascending order) and **DO.** (descending order) in the appropriate columns. When specifying orders on multiple columns, a number in parentheses after **AO.** or **DO.** specifies the precedence of the columns.

Example 15.55 The following query lists the *instock* relation by decreasing quantity and increasing part number.

instock	PART#	LOCATION	QUANTITY
P.	**AO**(2)**.**		**DO**(1)**.**

Arithmetic expressions may appear in QBE skeletons and the condition box.

Example 15.56 Which locations have at least half the number of a part used on a given aircraft?

usedon	PART#	PTYPE	NUSED
	$\underline{100}$	$\underline{\text{DC10}}$	$\underline{25}$

instock	PART#	LOCATION	QUANTITY
	$\underline{100}$	$\underline{\text{Chicago}}$	$>= 0.5 * \underline{25}$

hashalf	PART#	PTYPE	LOCATION
P.	100	DC10	Chicago

Example 15.57 What proportion of the number of coach seats used on a 707 does each location have?

pinfo	PART#	SUBPARTOF	PARTNAME
	100		coach seat

usedon	PART#	PTYPE	NUSED
	100	707	25

instock	PART#	LOCATION	QUANTITY
	100	Chicago	30

seats	LOCATION	PROPORTION
P.	Chicago	30/25

QBE has the aggregate operators **CNT.**, **SUM.**, **AVG.**, **MAX.**, and **MIN.** that can be applied to an entry in a row. The entry must be prefixed with **ALL.** to indicate that all values for the entry are to be collected and treated as a set.

Example 15.58 How many of part 211 are in stock?

instock	PART#	LOCATION	QUANTITY
	211		**P. SUM. ALL.** 50

Result:

instock	QUANTITY Sum
	211

QBE appends the word "Sum" (and appropriate words for other aggregate operators) to the column heading to indicate that the value in the result is an aggregate rather than a directly-retrieved value.

To eliminate duplicates in a set formed by **ALL.**, the operator **UNQ.** is used.

Example 15.59 How many locations have parts?

instock	PART#	LOCATION	QUANTITY
		P. CNT. UNQ. ALL.	

Grouping before application of aggregate operators is accomplished by the operator **G.** in the columns on which the grouping is to take place.

Example 15.60 Which parts have the number needed on a 727 in at least one location?

usedon	PART#	PTYPE	NUSED
	$\underline{100}$	727	$\underline{50}$

instock	PART#	LOCATION	QUANTITY
	P. G. $\underline{100}$		**MAX. ALL.** $\underline{25}$

CONDITIONS
MAX. ALL. $\underline{25} >= \underline{50}$

Result:

instock	PART#
	2116 21164

Example 15.61 How many of each part are there?

instock	PART#	LOCATION	QUANTITY
	P. G.		**P. SUM. ALL.**

Example 15.62 At how many locations is each part stocked?

instock	PART#	LOCATION	QUANTITY
	P. G.	**P. CNT. ALL.**	

Result:

instock	PART#	LOCATION Count
	211	3
	2114	2
	2116	2
	21164	1

We look finally at a feature of QBE, access to the transitive closure of a relation, that is unique among relational query languages. (However, the feature is not available in current implementations.) The transitive closure mechanism assumes a pair of attributes in a relation for which the transitive closure of the projection of the relation on those two attributes is tree-structured. *PART#* and *SUBPARTOF* in *pinfo* form such a pair. No part is a subpart of itself, at any level, and no part is a subpart of more than one part. (The last restriction may not seem likely in general, but it holds in the current state of *pinfo*.) The structure described by the transitive closure on *PART#* and *SUBPARTOF* in *pinfo* is shown in Figure 15.2.

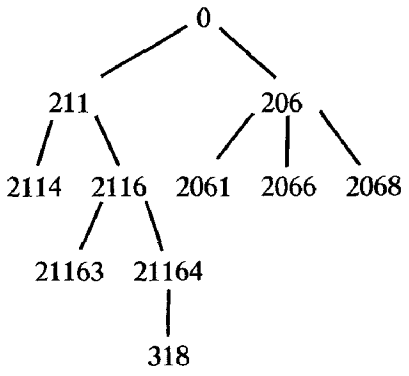

Figure 15.2

If we wanted to answer the question "What are the subparts of subparts of part 211?", we could use the following QBE query.

pinfo	PART#	SUBPARTOF
	$\underline{100}$	211
	P. $\underline{101}$	$\underline{100}$

(QBE queries need not list all the attributes in a relation.) However, with the features of QBE outlined so far, there is no query to answer the question "What parts are subparts of part 211 at all levels?", unless there is an a priori bound on the number of levels of subparts there can be.

QBE allows direct reference to the transitive closure of a projection of a relation through the notation (*n*L) after an entry. The *n* can be either a positive integer constant or a variable. If *n* is a constant, it refers to the number of levels up or down the tree to go.

Example 15.63 What are the subparts of subparts of part 211?

pinfo	PART#	SUBPARTOF
	P. $\underline{100}$**(2L)**	211

Result:

pinfo	PART#

	2114(2L)
	2116(2L)

Note that level information is incorporated in the answer to the query.

Example 15.64 What part contains part 318 as a third-level subpart?

pinfo	PART#	SUBPARTOF
	318	**P.** $\underline{200}$(3L)

pinfo	SUBPARTOF
	211(3L)

The level operator, **L**, may be preceded by a variable to indicate a search up or down the tree by an indeterminate number of levels.

Example 15.65 At what level is part 21164 a subpart of part 211?

pinfo	PART#	SUBPARTOF
	P. 21164($\underline{4}$L)	211

Result:

pinfo	PART#
	21164(2L)

Example 15.66 What are the subparts of part 211 at all levels?

pinfo	PART#	SUBPARTOF
	P. 100(4**L**)	211

Result:

pinfo	PART#
	2114(1**L**)
	2116(1**L**)
	21163(2**L**)
	21164(2**L**)
	318(3**L**)

There are two operators, **MAX.** and **LAST.**, that can be used with **L** to refer to items at the lowest level in the tree and items that are leaves in the tree, respectively.

Example 15.67 What are the lowest level subparts?

pinfo	PART#	SUBPARTOF
	P. 100(**MAX. L**)	0

Result:

pinfo	PART#
	318(4**L**)

Example 15.68 What subparts of part 211, at any level, themselves have no subparts?

pinfo	PART#	SUBPARTOF
	P. <u>100</u>**(LAST. L)**	211

Result:

pinfo	PART#

	2114(1L)
	21163(2L)
	318(3L)

15.5 PIQUE

PIQUE (PIts QUEry language) is an experimental data retrieval language for the PITS (Pie-In-The-Sky) database system under development at the State University of New York at Stony Brook and the Oregon Graduate Center. PIQUE has a QUEL-like syntax. However, tuple variables are implicitly bound to windows, rather than explicitly bound to relations, thus removing range statements and join conditions. A QUEL query to answer the question "Which locations have as many coach seats as are used on a 707?" is

```
range of p is pinfo
range of u is usedon
range of i is instock
retrieve (i.LOCATION)
   where p.PART# = u.PART# and p.PART# = i.PART# and
      u.PTYPE = "707" and u.NUSED <= i.QUANTITY
```

The same question can be answered in PIQUE with

```
retrieve LOCATION where (PTYPE = "707") *
   (NUSED <= QUANTITY).
```

The semantics of PIQUE is defined relative to some window function. The only assumption PIQUE makes is that the window function obeys the containment condition. For examples in this section, we use an object-based window function $[]_{\mathbf{R},\mathbf{O}}$, where

$$\mathbf{R} = \{\mathit{PART\#\ SUBPARTOF},\ \mathit{PART\#\ PARTNAME},\ \mathit{PART\#\ PTYPE\ USEDON},\ \mathit{PART\#\ LOCATION\ NUSED}\}$$

and **O** consists of all nonempty unions of schemes in **R**, as shown in Figure 15.3.

O = **R** ∪
{*PART# SUBPARTOF PARTNAME,*
PART# SUBPARTOF PTYPE NUSED,
PART# PARTNAME PTYPE NUSED,
PART# SUBPARTOF PARTNAME PTYPE NUSED,
PART# SUBPARTOF LOCATION QUANTITY,
PART# PARTNAME LOCATION QUANTITY,
PART# SUBPARTOF PARTNAME LOCATION QUANTITY,
PART# PTYPE NUSED LOCATION QUANTITY,
PART# SUBPARTOF PTYPE NUSED LOCATION QUANTITY,
PART# PARTNAME PTYPE NUSED LOCATION QUANTITY,
PART# SUBPARTOF PARTNAME PTYPE NUSED LOCATION QUANTITY}

Figure 15.3

As we see from the objects, all two-, three-, and four-way joins of relations are permitted. For a database on **R** we use *usedon* and *instock* for *PART# PTYPE NUSED* and *PART# LOCATION QUANTITY*. For *PART# SUBPARTOF* and *PART# PARTNAME* we use

$$pinfo1 = \pi_{PART\#\ SUBPARTOF}(\sigma_{SUBPARTOF \neq 0}(pinfo))$$

and

$$pinfo2 = \pi_{PART\#\ PARTNAME}(pinfo).$$

The relations *pinfo*1 and *pinfo*2 are shown in Figure 15.4. By splitting *pinfo* in this way, we avoid using a special value for *SUBPARTOF* when a part is not a subpart of any part. We could have made the same decomposition for previous examples, but at the expense of complicating some queries. In PIQUE, there is no added complexity in queries for making this decomposition.

*pinfo*1

PART#	SUBPARTOF
2114	211
2116	211
21163	2116
21164	2116
318	21164
2061	206
2066	206
2068	206

*pinfo*2

PART#	PARTNAME
211	coach seat
2114	seat cover
2116	seat belt
21163	seat belt buckle
21164	seat belt anchor
318	funny little bolt
206	overhead console
2061	paging switch
2066	light switch
2068	air nozzle

Figure 15.4

For each tuple variable x in a PIQUE query, the *mention set* of x, denoted $men(x)$, is the set of attributes that appear with x in the query. In evaluating the query, x is bound to $[men(x)]$. Thus, if x appears with *PARTNAME* and *LOCATION*, it will be bound to $[PARTNAME\ LOCATION]$, which is

$$\pi_{PARTNAME\ LOCATION}(pinfo2 \bowtie instock).$$

The simplest PIQUE queries have a retrieve list and a sequence of selection conditions connected with *'s.

Example 15.69 Which locations have coach seats?

retrieve *x.LOCATION* **where** (*x.PARTNAME* = "coach seat")

(LOCATION)
JFK
Boston
O'Hare

Example 15.70 Which locations have at least as many of a part as are used on a 707?

retrieve *x.PART#, x.LOCATION* **where** (*PTYPE* = "707")
  * (*x.QUANTITY* > = *x.NUSED*)

Here x is bound to [*PART# LOCATION PTYPE QUANTITY*].

Example 15.71 Which parts are subparts of the same part?

retrieve *x.PART#* — > *SUBPART1*, *y.PART#* — > *SUBPART2*
  where (*x.SUBPARTOF* = *y.SUBPARTOF*)
    * (*x.PART#* < *y.PART#*)

Here x and y are bound to separate copies of [*PART# SUBPARTOF*]. The symbol — > is used for renaming attributes.

(SUBPART1	SUBPART2)
2114	2116
21163	21164
2061	2066
2061	2068
2066	2068

A syntactic simplification allowed in PIQUE is the use of a "blank" tuple variable. Any attributes not preceded by a tuple variable are assumed to be qualified by a special tuple variable "blank."

Example 15.72 The queries in the last three examples can be written as

> **retrieve** *LOCATION* **where** (*PARTNAME* = "coach seat")

> **retrieve** *PART#, LOCATION* **where** (*PTYPE* = "707")
>   * (*QUANTITY* >= *NUSED*)

> **retrieve** *PART#* −> *SUBPART*1, *y.PART#* −> *SUBPART*2
>   **where** (*SUBPARTOF* = *y.SUBPARTOF*) * (*PART#* < *y.PART#*)

In the last query, one explicit tuple variable is still needed.

The types of selection conditions seen so far are *simple conditions*: sequences of comparisons connected with *'s. *Compound conditions* are formed from simple conditions with the connectives **and**, **or**, and **not**. The semantics of queries with compound conditions is given by the following equivalences.

> **retrieve** ⟨list⟩ **where** ⟨condition1⟩ **and** ⟨condition2⟩ ≡
>   (**retrieve** ⟨list⟩ **where** ⟨condition1⟩) ∩
>   (**retrieve** ⟨list⟩ **where** ⟨condition2⟩)

> **retrieve** ⟨list⟩ **where** ⟨condition1⟩ **or** ⟨condition2⟩ ≡
>   (**retrieve** ⟨list⟩ **where** ⟨condition1⟩) ∪
>   (**retrieve** ⟨list⟩ **where** ⟨condition2⟩)

> **retrieve** ⟨list⟩ **where not** ⟨condition⟩ ≡
>   (**retrieve** ⟨list⟩) − (**retrieve** ⟨list⟩ **where** ⟨condition⟩)

Example 15.73 Which parts are in stock at both Boston and O'Hare?

> **retrieve** *PART#* **where** (*LOCATION* = "Boston") **and**
>   (*LOCATION* = "O'Hare")

(PART#)
211
2116

Example 15.74 Which parts are not in stock at Boston?

> **retrieve** *PART#* **where not** (*LOCATION* = "Boston")

(PART#)
2114
21163
21164
318
206
2061
2066
2068

The difference between * and **and** is important. Using * means that the conditions connected will be enforced on the same window, while **and** enforces the conditions on two windows and intersects the results, possible after projection. The connectives **and**, **or**, and **not** are essentially shorthand for writing certain two-variable queries with one variable. Referring back to Example 15.73, if we instead wrote

> **retrieve** *PART#* **where** (*LOCATION* = "Boston")
> * (*LOCATION* = "O'Hare"),

the result would be an empty relation, since no tuple t has both t(*LOCATION*) = Boston and t(*LOCATION*) = O'Hare. Another consideration is that **and** can be used when no object spans all the attributes mentioned.

Example 15.75 For this example, assume that all objects that mention both *PTYPE* and *LOCATION* are removed from **O**. The query

> **retrieve** *PART#* **where** (*PTYPE* = "707")
> * (*LOCATION* = "Boston")

cannot be used, while

> **retrieve** *PART#* **where** (*PTYPE* = "707") **and**
> (*LOCATION* = "Boston")

would work.

Another important distinction is between a negated condition and the condition with the comparator complemented, such as **not** (*LOCATION* = "Boston") and (*LOCATION* ≠ "Boston").

Example 15.76 The query

> **retrieve** *PART#* **where not** (*LOCATION* = "Boston")

from Example 15.74 is asking for any part not stocked at Boston, while

> **retrieve** *PART#* **where** (*LOCATION* ≠ "Boston")

asks for parts stocked at some location besides Boston. The answer to the second query is

(PART#)
211
2114
2116
21164

To get parts not stocked at Boston, but stocked at some location, we write

> **retrieve** *PART#*
> **where** (*LOCATION* = *LOCATION*) **and not**
> (*LOCATION* = "Boston")

(PART#)
2114
21164

The last query in the preceding example contains the condition (*LOCATION* = *LOCATION*). At first sight, that condition may seem superfluous, since it is always true. However, it affects the answer to the query, since it changes the mention set of the tuple variable. A condition such as (*LOCATION* = *LOCATION*) can be abbreviated to (*LOCATION*), and is called a *name drop*. The purpose of a name drop is to include an attribute in the realm of discourse, with no condition other than that it be present.

Example 15.77 What parts are there?

retrieve *PART#*

```
(PART#)
  211
 2114
 2116
21163
21164
  318
  206
 2061
 2066
 2068
```

Example 15.78 Which parts are stocked at some location?

retrieve *PART#* **where** *LOCATION*

```
(PART#)
  211
 2114
 2116
21164
```

Example 15.79 Which parts are not subparts of any part?

retrieve *PART#* **where not** (*SUBPARTOF*)

```
(PART#)
  211
  206
```

PIQUE allows references to subqueries via the keyword **in**, which specifies a semijoin of the window for a tuple variable with the result of the subquery. Thus, **in** denotes set membership of a tuple (or parts of it) in the set of tuples resulting from the subquery. The tuple variables in the subquery are local to the subquery, and are not connected with tuple variables by the same name in the containing query.

Example 15.80 Which locations stock some part that Boston stocks?

```
retrieve PART#, LOCATION
  where (LOCATION ≠ "Boston") * (PART# in
    retrieve PART# where (LOCATION = "Boston"))
```

(PART#	LOCATION)
211	JFK
211	O'Hare
2116	O'Hare

15.6 BIBLIOGRAPHY AND COMMENTS

ISBL and the PRTV system are described by Todd [1975, 1976]. Stonebraker, Wong, *et al.* [1976] present QUEL and the INGRES system. Astrahan, Blasgen, *et al.* [1976, 1980], Blasgen, Astrahan, *et al.* [1981] and Chamberlin, Astrahan, *et al.* [1981] report on System R. Boyce, Chamberlin, *et al.* [1975] introduce SQUARE and an early version of SQL. SQL is covered in more detail by Astrahan and Chamberlin [1975] and Astrahan, Chamberlin, *et al.* [1976]. SEQUEL and SEQUEL2 are names of earlier versions of SQL. QBE is described in a series of papers by Zloof [1976, 1977, 1981]. PIQUE is presented by Maier, Rozenshtein, *et al.* [1981]. PIQUE is based heavily upon a similar language, System/U, under development at Stanford. Korth [1981] and Korth and Ullman [1980] describe that language.

Chamberlin [1976] and Kim [1979] both give surveys of relational database management systems. Pirotte [1979] classifies relational query languages by whether their underlying structure is relational algebra, tuple calculus, or domain calculus. Cooper [1980] compares the expressive power of various relational query languages.