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 lan-
guages 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,

\[ r - s = r - (r \bowtie s), \]

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, \( A_1 \rightarrow A_2 \) 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
used to stand for all unnamed attributes in a relation. Thus, if we want to rename just B in r, we would write $r \% B \rightarrow D$, ... 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 state-
ment, with the result of evaluating the query included sometimes.

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

\[
\text{list } \textit{pinfo} : \text{\texttt{PART\#}} = 2116 \% \text{\texttt{PARTNAME}} \\
\text{(PARTNAME)} \\
\text{seat belt buckle} \\
\text{seat belt anchor}
\]

Example 15.2  How many seat belts are at Boston?

\[
\text{list } \textit{pinfo} \ast \textit{instock} : \text{\texttt{PARTNAME}} = \text{"seat belt"} \& \\
\text{\texttt{LOCATION}} = \text{"Boston"}\% \text{\texttt{QUANTITY}} \\
\text{(QUANTITY)} \\
341
\]

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

\[
\text{list } \textit{pinfo} \sim (\textit{instock} : \text{\texttt{LOCATION}} = \text{"Boston"}\mid \\
\text{\texttt{LOCATION}} = \text{"O'Hare"})\% \text{\texttt{PART\#}} \\
\text{(PART\#)} \\
21163 \\
21164 \\
318 \\
206 \\
2061 \\
2066 \\
2068
\]

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

\[
\text{list } \textit{pinfo} \% \text{\texttt{PART\#}} \rightarrow \text{\texttt{SUBSUBPART}}, \text{\texttt{SUBPARTOF}} \rightarrow \text{\texttt{PART\#}}\* \\
\textit{pinfo} : \text{\texttt{PARTNAME}} = \text{"seat belt"} \% \text{\texttt{SUBSUBPART}} \\
\text{(SUBSUBPART)} \\
21163 \\
21164
\]
Example 15.5 Which parts used on a 707 are in stock at Atlanta?

\[
\text{list usedon : PTYPE = "707" \% PART#} \\
(\text{instock : LOCATION = "Atlanta" \% PART#}) \\
\underline{(PART#)} \\
21164
\]

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

\[
\begin{align*}
\text{r1} &= \text{usedon : PTYPE = "707" \% PART#} \\
\text{r2} &= \text{instock : LOCATION = "Atlanta" \% PART#} \\
\text{r3} &= \text{r1} \cdot \text{r2} \\
\text{list r3}
\end{align*}
\]

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 \text{stock707} that gives part number, number used, location, and quantity for all parts used in a 707. If we make the assignment

\[
\text{stock707} = \text{N!usedon} \ast \text{N!instock : PTYPE = "707" \% PART#, NUSED, LOCATION, QUANTITY}
\]

then any time we write \text{list stock707} or \text{stock707} occurs in an expression being evaluated, the current values of \text{usedon} and \text{instock} are used. If the N! is removed in both places, \text{stock707}'s value will depend on the state of \text{usedon} and \text{instock} at the point when \text{stock707} 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 $\text{adequate}(a, b)$ be a function that returns $true$ exactly when $a \geq 0.5b$. The ISBL statement

\[
\text{list usedon * instock * adequate(QUANTITY, NUSED) \%}
\text{PART#, PTYPE, LOCATION}
\]

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

<table>
<thead>
<tr>
<th>(PART#</th>
<th>PTYPE</th>
<th>LOCATION)</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td>707</td>
<td>JFK</td>
</tr>
<tr>
<td>211</td>
<td>727</td>
<td>JFK</td>
</tr>
<tr>
<td>211</td>
<td>707</td>
<td>O'Hare</td>
</tr>
<tr>
<td>211</td>
<td>727</td>
<td>O'Hare</td>
</tr>
<tr>
<td>2116</td>
<td>707</td>
<td>Boston</td>
</tr>
<tr>
<td>2116</td>
<td>727</td>
<td>Boston</td>
</tr>
<tr>
<td>21164</td>
<td>707</td>
<td>Atlanta</td>
</tr>
<tr>
<td>21164</td>
<td>727</td>
<td>Atlanta</td>
</tr>
</tbody>
</table>

**Example 15.8** Let $\text{needed}(a, b, c)$ be a procedure that takes $a$ and $b$ as input and returns $c = \max(0.5b - a, 0)$. The statement

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

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):

<table>
<thead>
<tr>
<th>(PART#</th>
<th>LOCATION</th>
<th>ORDER)</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td>Boston</td>
<td>39</td>
</tr>
<tr>
<td>2114</td>
<td>JFK</td>
<td>61</td>
</tr>
<tr>
<td>2114</td>
<td>O'Hare</td>
<td>39</td>
</tr>
<tr>
<td>2116</td>
<td>O'Hare</td>
<td>119</td>
</tr>
</tbody>
</table>
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,

\[
\text{range of (tuple variable) is (relation name).}
\]

The A-component of a tuple variable \( x \) is denoted as \( x.A \).

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

\begin{align*}
\text{range of } p & \text{ is } pinfo \\
\text{range of } p_1 & \text{ is } pinfo \\
\text{range of } u & \text{ is } usedon \\
\text{range of } i & \text{ is } instock
\end{align*}

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

\[
\text{retrieve ((target list)) where (condition).}
\]

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?

```sql
retrieve (p.PARTNAME) where p.PART# = 2116
```

(PARTNAME)

seat belt

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

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

(PARTNAME QUANTITY)

<table>
<thead>
<tr>
<th>PARTNAME</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>coach seat</td>
<td>77</td>
</tr>
<tr>
<td>seat cover</td>
<td>28</td>
</tr>
<tr>
<td>seat belt</td>
<td>29</td>
</tr>
</tbody>
</table>

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?

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

(SUBPART1 SUBPART2)

<table>
<thead>
<tr>
<th>SUBPART1</th>
<th>SUBPART2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2114</td>
<td>2116</td>
</tr>
<tr>
<td>21163</td>
<td>21164</td>
</tr>
<tr>
<td>2061</td>
<td>2066</td>
</tr>
<tr>
<td>2061</td>
<td>2068</td>
</tr>
<tr>
<td>2066</td>
<td>2068</td>
</tr>
</tbody>
</table>

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"
```

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>PROPORTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFK</td>
<td>1.233</td>
</tr>
<tr>
<td>Boston</td>
<td>.326</td>
</tr>
<tr>
<td>O'Hare</td>
<td>.894</td>
</tr>
</tbody>
</table>

Assignment of the result of a retrieval to a relation is achieved via the notation into (relation) after retrieve.

**Example 15.15**

```
retrieve into used727 (u.all) where u.PTYPE = "727"
```

makes used727 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#))
```

(\text{NUMPARTS})

\begin{tabular}{cc}
10
\end{tabular}

**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)
```

\begin{tabular}{cc}
(PART# & LOCATION) \\
2114 & JFK \\
21164 & Atlanta
\end{tabular}

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))
```

(\text{NUMLOCS})

\begin{tabular}{c}
4
\end{tabular}

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?

\[
\text{retrieve} \ (TOTAL = \text{sum}(i.QUANTITY \text{ where } i.PART# = 211))
\]

| (TOTAL) | 211 |

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

\[
\text{retrieve} \ (NUMBELTS = \text{sum}(i.QUANTITY))
\text{ where } p.PART# = i.PART# \text{ and } p.PARTNAME = \text{"seat belt"}
\]

The \text{sum} is computed independently of the rest of the query.

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

Example 15.21  How many of each part are there?

\[
\text{retrieve} \ (i.PART#, \ TOTAL = \text{sum}(i.QUANTITY \text{ by } i.PART#))
\]

<table>
<thead>
<tr>
<th>(PART#  TOTAL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>211 211</td>
</tr>
<tr>
<td>2114 34</td>
</tr>
<tr>
<td>2116 370</td>
</tr>
<tr>
<td>21164 36,391</td>
</tr>
</tbody>
</table>

Example 15.22  How many seat belts are there in stock?

\[
\text{retrieve} \ (NUMBELTS = \text{sum}(i.QUANTITY \text{ by } i.PART#))
\text{ where } p.PART# = i.PART# \text{ and } p.PARTNAME = \text{"seat belt"}
\]

| (NUMBELTS) | 370 |

Example 15.23  How many of each part over the maximum needed by any aircraft does each location have in stock?
retrieve \((u.PART#, \ i.LOLOCATION, OVERSTOCK = (i.QUANTITY - \max(u.NUSED \ by \ u.PART#)))\)
where \(u.PART# = i.PART# \text{ and} (i.QUANTITY - \max(u.NUSED \ by \ u.PART#)) > 0\)

\[
\begin{array}{ccc}
\text{PART#} & \text{LOCATION} & \text{OVERSTOCK} \\
2116 & \text{Boston} & 45 \\
21164 & \text{Atlanta} & 35,799 \\
\end{array}
\]

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

\[
\text{select } \langle \text{attribute list} \rangle \\
\text{from } \langle \text{relation} \rangle \\
\text{where } \langle \text{condition} \rangle \\
\]

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

\[
\text{select } \text{PARTNAME, PART#} \\
\text{from } \text{pinfo} \\
\text{where } \text{SUBPARTOF} = 211 \\
\]

\[
\begin{array}{ll}
\text{PARTNAME} & \text{PART#} \\
\text{seat cover} & 2114 \\
\text{seat belt} & 2116 \\
\end{array}
\]
Example 15.25  What parts are at Boston or O'Hare?

```sql
select PART#
from instock
where LOCATION = "Boston" or LOCATION = "O'Hare"
```

(The 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

```sql
select unique PART#
from instock
where LOCATION = "Boston" or LOCATION = "O'Hare"
```

(The 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?

```sql
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 irzstock
    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 $\text{set}(\text{PART#})$ refers to the set of all $\text{PART#}$-values occurring with a $\text{LOCATION}$-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?

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

Note that $\text{QUANTITY}$ refers to $\text{instock}$.

- (PART# LOCATION)
  - 211 JFK
  - 2116 Boston
  - 21164 Atlanta

Another formulation of this question is

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2114</td>
<td>2116</td>
</tr>
<tr>
<td>21163</td>
<td>21164</td>
</tr>
<tr>
<td>2061</td>
<td>2066</td>
</tr>
<tr>
<td>2061</td>
<td>2068</td>
</tr>
<tr>
<td>2066</td>
<td>2068</td>
</tr>
</tbody>
</table>

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
```
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?

```sql
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?

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

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?

\[
\text{select } \text{PART#}, \text{sum(QUANTITY)} \\
\text{from instock} \\
\text{group by PART#}
\]

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?

\[
\text{select usedon.PART#, NUSED - sum(QUANTITY)} \\
\text{from usedon, instock} \\
\text{where usedon.PART# = instock.PART# and PTYPE = "707"} \\
\text{group by usedon.PART#} \\
\text{having NUSED - sum(QUANTITY) } \geq 0
\]

<table>
<thead>
<tr>
<th>(PART#     )</th>
</tr>
</thead>
<tbody>
<tr>
<td>211  125</td>
</tr>
<tr>
<td>2116  136</td>
</tr>
<tr>
<td>21164 35,903</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
<th>PARTNAME</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>PART#</th>
<th>SUBPARTOF</th>
<th>PARTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>pinfo</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>211</td>
<td>P. Chicago</td>
<td>25</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>instock</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFK</td>
<td></td>
</tr>
<tr>
<td>Boston</td>
<td></td>
</tr>
<tr>
<td>O'Hare</td>
<td></td>
</tr>
</tbody>
</table>

Note that the name 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

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>211</td>
<td>P.</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P.</td>
<td></td>
<td>( \geq 50 )</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>2116</td>
</tr>
<tr>
<td></td>
<td>21164</td>
</tr>
</tbody>
</table>

Note that QBE does eliminate duplicates.

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

<table>
<thead>
<tr>
<th>usedon</th>
<th>PART#</th>
<th>PTYPE</th>
<th>NUSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td>( \neg = 727 )</td>
<td>( \geq 100 )</td>
<td></td>
</tr>
</tbody>
</table>
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?

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 100</td>
<td>100</td>
<td>Boston</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>O'Hare</td>
<td></td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>2116</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 100</td>
<td>100</td>
<td>Boston</td>
<td></td>
</tr>
<tr>
<td>P. 101</td>
<td>101</td>
<td>O'Hare</td>
<td></td>
</tr>
</tbody>
</table>
Example 15.47 The following query retrieves information on part 211.

<table>
<thead>
<tr>
<th>in stock</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>2114</td>
</tr>
<tr>
<td></td>
<td>2116</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>info</th>
<th>PART#</th>
<th>SUBPARTOF</th>
<th>PARTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td>211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P. 211</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P. 211</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>info</th>
<th>PART#</th>
<th>SUBPARTOF</th>
<th>PARTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 211</td>
<td></td>
<td>211</td>
<td>seat belt</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 100</td>
<td>P. Chicago</td>
<td>Boston</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>2114</td>
<td>JFK</td>
<td></td>
</tr>
<tr>
<td>2114</td>
<td>O'Hare</td>
<td></td>
</tr>
<tr>
<td>21164</td>
<td>Atlanta</td>
<td></td>
</tr>
</tbody>
</table>

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

**Example 15.50** The query

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
<th>PARTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td></td>
<td>P. seatbelt</td>
</tr>
</tbody>
</table>

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

Result:

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>PARTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2114</td>
<td></td>
<td>seat cover</td>
</tr>
<tr>
<td>2116</td>
<td></td>
<td>seat belt</td>
</tr>
<tr>
<td>21163</td>
<td></td>
<td>seat belt buckle</td>
</tr>
<tr>
<td>21164</td>
<td></td>
<td>seat belt anchor</td>
</tr>
</tbody>
</table>
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?

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
<th>PARTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>P. bolt</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>usedon</th>
<th>PART#</th>
<th>PTYPE</th>
<th>NUSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>707</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 100</td>
<td>JFK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 100</td>
<td>Chicago</td>
<td>&gt;= 50</td>
<td></td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
<th>PARTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>bolt</td>
</tr>
</tbody>
</table>
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?
The order of tuples in a result can be controlled by the prefixes \texttt{AO}. (ascending order) and \texttt{DO}. (descending order) in the appropriate columns. When specifying orders on multiple columns, a number in parentheses after \texttt{AO}. or \texttt{DO}. specifies the precedence of the columns.

\textbf{Example 15.55} The following query lists the \textit{instock} relation by decreasing quantity and increasing part number.

\begin{verbatim}

\begin{tabular}{l|c|c}
\hline
\textit{instock} & \texttt{PART#} & \texttt{LOCATION} \\
\hline
\texttt{P.} & \texttt{AO(2).} & \texttt{DO(1).} \\
\hline
\end{tabular}

\end{verbatim}

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

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

\begin{verbatim}

\begin{tabular}{c|c|c|c}
\hline
\texttt{usedon} & \texttt{PART#} & \texttt{PTYPE} & \texttt{NUSED} \\
\hline
100 & DC10 & 25 \\
\hline
\end{tabular}

\begin{tabular}{c|c|c|c}
\hline
\texttt{instock} & \texttt{PART#} & \texttt{LOCATION} & \texttt{QUANTITY} \\
\hline
100 & Chicago & \texttt{\geq 0.5 \times 25} \\
\hline
\end{tabular}

\end{verbatim}
Example 15.57  What proportion of the number of coach seats used on a 707 does each location have?

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
<th>PARTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>coach seat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>usedon</th>
<th>PART#</th>
<th>PTYPE</th>
<th>NUSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>707</td>
<td></td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Chicago</td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>seats</th>
<th>LOCATION</th>
<th>PROPORTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td>Chicago</td>
<td>30 / 25</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td></td>
<td>P. SUM. ALL.</td>
<td>50</td>
</tr>
</tbody>
</table>
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?

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P. CNT. UNQ. ALL.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>usedon</th>
<th>PART#</th>
<th>PTYPE</th>
<th>NUSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>727</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. G. 100</td>
<td></td>
<td>MAX. ALL. 25</td>
<td></td>
</tr>
</tbody>
</table>

**CONDITIONS**

MAX. ALL. 25 ≥ 50
### Example 15.61
How many of each part are there?

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2116</td>
</tr>
<tr>
<td></td>
<td>21164</td>
</tr>
</tbody>
</table>

### Example 15.62
At how many locations is each part stocked?

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P. G.</td>
<td>P. SUM. ALL.</td>
<td></td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>instock</th>
<th>PART#</th>
<th>LOCATION</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>211</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2114</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2116</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>21164</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

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 sub-part 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
The structure described by the transitive closure on PART# and SUBPARTOF in pinfo is shown in Figure 15.2.

![Figure 15.2](image)

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

\[
\text{pinfo} \text{ PART# } \text{ SUBPARTOF} \\
\hline
\text{100} \quad 211 \\
\text{P. 101} \quad 100 \\
\hline
\]

(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 \((nL)\) 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?

\[
\text{pinfo} \text{ PART# } \text{ SUBPARTOF} \\
\hline
\text{P. 100(2L)} \quad 211 \\
\hline
\]
Result:

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2114(2L)</td>
</tr>
<tr>
<td></td>
<td>2116(2L)</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>P. 200(3L)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pinfo</th>
<th>SUBPARTOF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>211(3L)</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 21164(4L)</td>
<td>211</td>
<td></td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>21164(2L)</td>
</tr>
</tbody>
</table>
Example 15.66  What are the subparts of part 211 at all levels?

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 100(4L)</td>
<td>211</td>
<td></td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2114(1L)</td>
</tr>
<tr>
<td></td>
<td>2116(1L)</td>
</tr>
<tr>
<td></td>
<td>21163(2L)</td>
</tr>
<tr>
<td></td>
<td>21164(2L)</td>
</tr>
<tr>
<td></td>
<td>318(3L)</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
<th>SUBPARTOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>P. 100(MAX. L)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>pinfo</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>318(4L)</td>
</tr>
</tbody>
</table>

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

\[
\text{range of } p \text{ is } pinfo \\
\text{range of } u \text{ is } usedon \\
\text{range of } i \text{ is } instock \\
\text{retrieve } (i.\text{LOCATION}) \\
\text{where } p.\text{PART}\# = u.\text{PART}\# \text{ and } p.\text{PART}\# = i.\text{PART}\# \text{ and } u.\text{PTYPE} = "707" \text{ and } u.\text{NUSED} \leq i.\text{QUANTITY}
\]

The same question can be answered in PIQUE with

\[
\text{retrieve LOCATION where } (\text{PTYPE} = "707") * \\
(\text{NUSED} \leq \text{QUANTITY}).
\]

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

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

and $\mathbf{O}$ consists of all nonempty unions of schemes in $\mathbf{R}$, as shown in Figure 15.3.

$$
\mathbf{O} = \mathbf{R} \cup \\
\{\text{PART# SUBPARTOF PARTNAME, PART# SUBPARTOF PTYPE NUSED, PART# PARTNAME PTYPE NUSED, PART# SUBPARTOF PARTNAME PTYPE NUSED, PART# SUBPARTOF LOCATION QUANTITY, PART# PARTNAME LOCATION QUANTITY, PART# SUBPARTOF PARTNAME LOCATION QUANTITY, PART# PTYPE NUSED LOCATION QUANTITY, PART# SUBPARTOF PTYPE NUSED LOCATION QUANTITY, PART# PARTNAME PTYPE NUSED LOCATION QUANTITY, PART# SUBPARTOF PARTNAME PTYPE NUSED LOCATION QUANTITY} \}
$$

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

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

and

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

The relations $pinfo1$ and $pinfo2$ 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.

\[
\text{pinfo1(PART# SUBPARTOF)} \\
2114 & 211 \\
2116 & 211 \\
21163 & 2116 \\
21164 & 2116 \\
318 & 21164 \\
2061 & 206 \\
2066 & 206 \\
2068 & 206 \\
\]

\[
\text{pinfo2(PART# PARTNAME)} \\
211 & \text{coach seat} \\
2114 & \text{seat cover} \\
2116 & \text{seat belt} \\
21163 & \text{seat belt buckle} \\
21164 & \text{seat belt anchor} \\
318 & \text{funny little bolt} \\
206 & \text{overhead console} \\
2061 & \text{paging switch} \\
2066 & \text{light switch} \\
2068 & \text{air nozzle} \\
\]

\text{Figure 15.4}

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

\[\pi_{\text{PARTNAME LOCATION}}(\text{pinfo2 } \bowtie \text{ instock}).\]

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?

\[
\text{retrieve } x.\text{LOCATION where } (x.\text{PARTNAME} = \text{"coach seat"})
\]

(LOCATION)

<table>
<thead>
<tr>
<th>JFK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
</tr>
<tr>
<td>O'Hare</td>
</tr>
</tbody>
</table>

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

\[
\text{retrieve } x.\text{PART#}, x.\text{LOCATION where } (\text{PTYPE} = \text{"707"})
\]
\[
* (x.\text{QUANTITY} \geq x.\text{NUSED})
\]

Here \(x\) is bound to \([\text{PART# LOCATION PTYPE QUANTITY}]\).

Example 15.71  Which parts are subparts of the same part?

\[
\text{retrieve } x.\text{PART#} \rightarrow \text{SUBPART1}, y.\text{PART#} \rightarrow \text{SUBPART2}
\]
\[
\text{where } (x.\text{SUBPARTOF} = y.\text{SUBPARTOF})
\]
\[
* (x.\text{PART#} < y.\text{PART#})
\]

Here \(x\) and \(y\) are bound to separate copies of \([\text{PART# SUBPARTOF}]\). The symbol \(\rightarrow\) is used for renaming attributes.

<table>
<thead>
<tr>
<th>SUBPART1</th>
<th>SUBPART2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2114</td>
<td>2116</td>
</tr>
<tr>
<td>21163</td>
<td>21164</td>
</tr>
<tr>
<td>2061</td>
<td>2066</td>
</tr>
<tr>
<td>2061</td>
<td>2068</td>
</tr>
<tr>
<td>2066</td>
<td>2068</td>
</tr>
</tbody>
</table>

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# -> SUBPART1, y.PART# -> SUBPART2
  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)) \cap
  (retrieve (list) where (condition2))
```

```
retrieve (list) where (condition1) or (condition2) ≡
  (retrieve (list) where (condition1)) \cup
  (retrieve (list) where (condition2))
```

```
retrieve (list) where not (condition) ≡
  (retrieve (list)) \cdot (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?

\[
\text{retrieve } \text{PART# where not } (\text{LOCATION} = \text{“Boston”})
\]

<table>
<thead>
<tr>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td>2114</td>
</tr>
<tr>
<td>21163</td>
</tr>
<tr>
<td>21164</td>
</tr>
<tr>
<td>318</td>
</tr>
<tr>
<td>206</td>
</tr>
<tr>
<td>2061</td>
</tr>
<tr>
<td>2066</td>
</tr>
<tr>
<td>2068</td>
</tr>
</tbody>
</table>

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

\[
\text{retrieve } \text{PART# where } (\text{LOCATION} = \text{“Boston”}) \* (\text{LOCATION} = \text{“O’Hare”}),
\]

the result would be an empty relation, since no tuple \(t\) has both \(t(\text{LOCATION}) = \text{Boston}\) and \(t(\text{LOCATION}) = \text{O’Hare}\). 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

\[
\text{retrieve } \text{PART# where } (\text{PTYPE} = \text{“707”}) \* (\text{LOCATION} = \text{“Boston”})
\]

cannot be used, while

\[
\text{retrieve } \text{PART# where } (\text{PTYPE} = \text{“707”}) \text{ and } (\text{LOCATION} = \text{“Boston”})
\]

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

**Example 15.76** The query

\[
\text{retrieve } PART\# \text{ where not } (LOCATION = "Boston")
\]

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

\[
\text{retrieve } PART\# \text{ where } (LOCATION \neq "Boston")
\]

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

\[
\begin{array}{c}
\text{(PART\#)} \\
211 \\
2114 \\
2116 \\
21164
\end{array}
\]

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

\[
\begin{array}{c}
\text{retrieve } PART\#
\end{array}
\]

\[
\text{where } (LOCATION = LOCATION) \text{ and not } (LOCATION = "Boston")
\]

\[
\begin{array}{c}
\text{(PART\#)} \\
2114 \\
21164
\end{array}
\]

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?

\textbf{retriever} \textit{PART#}

\begin{verbatim}
(PART#)
  211
  2114
  2116
  21163
  21164
  318
  206
  2061
  2066
  2068
\end{verbatim}

Example 15.78  Which parts are stocked at some location?

\textbf{retriever} \textit{PART# where LOCATION}

\begin{verbatim}
(PART#)
  211
  2114
  2116
  21164
\end{verbatim}

Example 15.79  Which parts are not subparts of any part?

\textbf{retriever} \textit{PART# where not (SUBPARTOF)}

\begin{verbatim}
(PART#)
  211
  206
\end{verbatim}

PIQUE allows references to subqueries via the keyword \textbf{in}, which specifies a \textit{semijoin of the window for a tuple variable with the result of the subquery}. Thus, \textbf{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"))
```

<table>
<thead>
<tr>
<th>PART#</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td>JFK</td>
</tr>
<tr>
<td>211</td>
<td>O'Hare</td>
</tr>
<tr>
<td>2116</td>
<td>O'Hare</td>
</tr>
</tbody>
</table>

15.6 BIBLIOGRAPHY AND COMMENTS
