Chapter 6
Relational Algebra

Relational algebra is the principle component of the manipulative part of the RDB. We will study it in detail in this chapter.

There are two groups of operators, four traditional set operators such as Cartesian Product, Union, Intersection, and Difference; together with four special relational operators: Restrict, Project, Join, and Divide.
Eight operators illustrated...
...and in words

*Product* returns a relation containing all possible tuples that are a combination of two tuples, one from each of the two specified relations.

*Union* returns a relation containing all tuples that appear in either, or both, of the two specified relations.

*Intersect* returns a relation containing all tuples that appear in both of the two specified relations.

*Difference* returns a relation containing all tuples that appear in the first but not the second of the two relations.
Restrict returns a relation containing all tuples from a specified relation that satisfy a specific condition.

Project returns a relation containing all (sub)tuples that remain in a specified relation after specified attributes have been removed.

Join returns a relation containing all possible tuples that are a combination of two tuples, one from each of the two specific relations, such that the two tuples contributing to any given combination have a common value for the common attributes of the two relations (and that a common value appears just once, not twice, in the result tuple).

Divide takes two unary relations and one binary one as its inputs. As the output, it sends back a relation containing all tuples from the first unary relation that appear in the binary relation, which is matched with all tuples in the second unary relation.
What is the heading?

When we use the above operators to form a relation, based on others, besides the body, we have to know what is the heading of the resulted relation. Otherwise, it would make no sense to write an expression such as (S JOIN P) WHERE CITY='Athens'; if we don’t know if S JOIN P has CITY as one of its attributes.

Thus, an relational expression should produce a result that, besides all the tuples, also has a well-defined relation type, including a well-defined set of attribute names.

We firstly introduce a new operator, RENAME, which is to rename attributes within a specified relation. E.g., the expression S RENAME CITY AS SCITY, although will not change the original rel-var, yields another relation, which is the same as S, except that the city attribute is (re)name as SCITY.
What should they look like?

Firstly, a relation expression should be constructed as follows:

\[
\text{<relational expression> ::= RELATION{<tuple expression>}| <relvar name> <relation operator> <relational expression>}
\]

E.g.,

\[
S := S \text{ UNION RELATION \{TUPLE \{S\# S\# ('S6'), SNAME NAME ('Smith'), STATUS 50, CITY 'Rome'}\};
\]

\[
\text{<relation operator> ::= <project> | <nonproject>}
\]
Continuing...

\[ \text{<project>::=}<\text{relational expression}>
\{[\text{ALL BUT }]<\text{attribute name commmalist}>\}\]

\[ \text{<nonproject>::=}<\text{rename}|<\text{union}|<\text{intersect}|<\text{minus}|<\text{ties}|<\text{restrict}|<\text{join}|<\text{divide}>\]

\[ \text{<rename>::=}<\text{relational expression}>
\text{RENAME}<\text{renaming commmalist}>\]

\[ \text{<union>::=}<\text{relational expression}>
\text{UNION}<\text{relational expression}>\]

The others are similar, and will be further demonstrated later.
What do they mean?

Given two relations $A$ and $B$ of the same type, the *union* of them, $A \text{ UNION } B$, is a relation of the same type, with body consisting of all tuples $t$ such that $t \in A$, or $t \in B$.

Given two relations $A$ and $B$ of the same type, the *intersect* of them, $A \text{ INTERSECT } B$, is a relation of the same type, with body consisting of all tuples $t$ such that $t \in A$, and $t \in B$.

Given two relations $A$ and $B$ of the same type, the *difference* between them, $A \text{ MINUS } B$, is a relation of the same type, with body consisting of all tuples $t$ such that $t \in A$, but $t \not\in B$. 
An example

Given the following two relations:

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
</tbody>
</table>

Then, below is the result of $A \cup B$.

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
</tbody>
</table>
Continuing...

Below is the result of $A$ INTERSECT $B$.

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
</tbody>
</table>

Finally, below is the result of $A$ MINUS $B$,

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
</tbody>
</table>

and the result of $B$ MINUS $A$.

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
</tbody>
</table>
Cartesian product of two relations $A$ and $B$, $A \times B$, where $A$ and $B$ share no common attributes names, is a relation with a heading that is the normal union of the headings of $A$ and $B$, with a body consisting of the set of all tuples $t$ such that $t = (a, b), a \in A$ and $t \in B$. For example, Let $A$ and $B$ two the following,

<table>
<thead>
<tr>
<th>S#</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P#</td>
<td>P1</td>
<td>P2</td>
</tr>
</tbody>
</table>

Then, the result of $A \times B$ is the following,

<table>
<thead>
<tr>
<th>S#</th>
<th>P#</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>P1</td>
</tr>
<tr>
<td>S1</td>
<td>P2</td>
</tr>
<tr>
<td>S2</td>
<td>P1</td>
</tr>
<tr>
<td>S2</td>
<td>P2</td>
</tr>
</tbody>
</table>
Restriction

Let $A$ be a relation with attributes $X$ and $Y$, and let $X \Theta Y$ be a well defined condition, then the $\Theta$–restriction of $A$ on $X$ and $Y$ is a relation with the same heading as $A$ and with body consisting of all tuples $t$ of $A$ such that the condition $X \Theta Y$ is true on $t$.

The involved condition can also be a more complex boolean condition. Then, we have the following:

\[ A \text{ WHERE } c_1 \text{ AND } c_2 \text{ means } (A \text{ WHERE } c_1) \text{ INTERSECT } (A \text{ WHERE } c_2). \]

\[ A \text{ WHERE } c_1 \text{ OR } c_2 \text{ means } (A \text{ WHERE } c_1) \text{ UNION } (A \text{ WHERE } c_2). \]

\[ A \text{ WHERE NOT } c \text{ means } A \text{ MINUS } (A \text{ WHERE } c). \]
An example

Given the following S relations:

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S5</td>
<td>Adams</td>
<td>30</td>
<td>Athens</td>
</tr>
</tbody>
</table>

Below is the result of “S WHERE CITY = ‘London’”:

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
</tbody>
</table>
Projection

Let $A$ be a relation with attributes $X_1, X_2, \ldots, X_n$. Then, the projection of $A$ on $X_1, X_2, \ldots, X_n$, $A\{X_1, X_2, \ldots, X_n\}$, is a relation, whose heading is derived from the heading of $A$ by removing all not mentioned in the set $\{X_1, X_2, \ldots, X_n\}$, and whose body consisting all tuples $\{x_1 : X_1, x_2 : X_2, \ldots, x_n : X_n\}$.

Questions: 1) Can we mention any attribute more than once?
2) What will be the result if every attribute name is mentioned in the list?
3) What happens if no attribute is named?
4) What does “$P \{\text{ALL BUT WEIGHT}\}$” mean?

Homework: Exercise 7.3.
An example

Given the following S relations:

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S5</td>
<td>Adams</td>
<td>30</td>
<td>Athens</td>
</tr>
</tbody>
</table>

Below is the result of "S WHERE CITY='Paris' {S#} ":

<table>
<thead>
<tr>
<th>S#</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
</tr>
</tbody>
</table>
Joins

Let relations $A$ and $B$ have headings $\{X_m, Y_n\}$ and $\{Y_n, Z_p\}$, respectively, i.e., they share the $Y$ attributes, and $X_m, Z_p$ are the other attributes of $A$ and $B$, respectively. Then, the natural join of $A$ and $B$, $A \text{ JOIN } B$, is a relation with heading $\{X_m, Y_n, Z_p\}$, and body consisting of the set of all tuples $\{x_m : X_m, y_n : Y_n, z_p : Z_p\}$, such that a tuple appears in $A$ with $(x_m, y_n)$, and another one in $B$ with $(y_n, z_p)$.

The above join can be regarded as an equality join. More generally, we can have $\Theta$–join of relation $A$ with attributes $X$ and $B$ with attributes $Y$ such that $X \Theta Y$, as a well-defined condition, is true. Such a joined relation is defined to be the result of evaluating the expression $(A \text{ TIMES } B) \text{ WHERE } X \Theta Y$. 
Examples

Assume that when “>” is applied to CITY, it means “greater in the alphabetic ordering”, then we can compute the greater-than join of relation S on CITY with relation P on CITY. The corresponding expression would be the follows:

$$(S \text{ RENAME CITY AS SCITY}) \times (P \text{ RENAME CITY AS PCITY}) \quad \text{WHERE } SCITY > PCITY$$

Thus, S JOIN P is equivalent to the following:

$$((S \times (P \text{ RENAME CITY AS PCITY})) \quad \text{WHERE CITY=PCITY}) \quad \{\text{ALL BUT PCITY}\}$$

**Homework:** Exercises 7.2 and 7.5.
Division

Let relations $A$ and $B$ have headings $\{X_m\}$ and $\{Y_n\}$, respectively, and let $C$ have heading $\{X_m, Y_n\}$. Then the division of $A$ by $B$ per $C$, $A$ DIVIDED BY $B$ PER $C$, is a relation with heading $X$ and body consisting of all tuples $\{x_m : X_m\}$ such that a tuple $\{x_m : X_m, y_n : Y_n\}$ appears in $C$ for all tuples $\{y_m : Y_m\}$ appearing in $B$. 
Associativity and commutativity

It is easy to verify that UNION is associative, i.e., \((A \cup B) \cup C\) and \(A \cup (B \cup C)\) are logically equivalent. Thus, we don’t need to add in any embedded parentheses. Similarly, INTERSECT, TIMES, and JOIN, are all associative. But MINUS is not.

It is also true that UNION is commutative, i.e. \(A \cup B\) is logically equivalent to \(B \cup A\). Similarly, INTERSECT, TIMES, and JOIN, are all commutative. Again, MINUS is not commutative.

**Homework:** Exercises 7.4 and 7.7.
From algebra to SQL

To construct an SQL query, we need to know *Select* what attributes of those rows *From* which tables *Where* certain restriction holds.

Such a query specifies *what* information to get, but not *how* to get it.

Although it is pretty easy to know what attributes to get, and to some degree, from what tables; it is usually not easy to specify the restrictions. This is where a relational algebraic expression can *often* guide us through.
Examples

1. Get supplier names for suppliers who supply part P2.

\[
((\text{SP JOIN } S) \text{ WHERE } P\#=P\#(\text{‘P2’})) \{\text{SNAME}\}
\]

It immediately converts to the following SQL query:

Select SNAME
From SP JOIN S
Where P\#=’P2’;

Since SQL does not directly support JOIN, we further put it into the following:

Select SNAME
From S, SP
Where S\#.S\#=SP\#.S\# AND SP\#.P\#=’P2’;
2. Get supplier names for suppliers who supply at least one red part.

$$(((P \text{ WHERE } \text{COLOR}='\text{Red}') \text{ JOIN } \text{SP})\{S\#\})$$

$$\text{JOIN } S)\{S\text{NAME}\}$$

Another form is the following:

$$(((S \text{ JOIN } \text{SP}) \text{ JOIN } P) \text{ WHERE } \text{COLOR}='\text{Red}')$$

$$\{S\text{NAME}\}$$

Similarly, the corresponding SQL query is the following:

Select SNAME
From S, P, SP
Where S.S#=SP.S# AND SP.P#=P.P#
    AND P.COLOR='Red';
3. Get supplier numbers for suppliers who either supplies a nut or a bolt.

We can use the UNION operation.

\[
((P \text{ Where } P\text{\textunderscore NAME}='\textbf{Nut}') \text{ JOIN } SP)\{S\#\} \text{ UNION } (P \text{ Where } P\text{\textunderscore NAME}='\textbf{Bolt}') \text{ JOIN } SP)\{S\#\}
\]

Its SQL correspondent simply the following:

```
Select S#
From SP, P
Where P.P#=SP.P# AND P.PNAME='Nut'
UNION
Select S#
From SP, P
Where P.P#=SP.P# AND P.PNAME='Bolt'
```
4. Get supplier names for those who don’t supply P2.

$$((S\{\text{SNAME}\} \text{ MINUS } (SP \text{ WHERE } P\#=P\#('P2'))\{S\#\}) \text{ JOIN } S) \{\text{SNAME}\}$$

Below is its conversion:

```sql
Select SNAME
from S
MINUS
Select SNAME
from S, SP
where S.S#=SP.S# and SP.P#='P2';
```
5. Get all pairs of supplier numbers such that the two suppliers concerned are located in the same city.

The word “same” suggests the JOIN operation, which is applied on two tables. Since we only want to have the same city, but not the same S#, we apply the RENAME operation to distinguish the S# of two instances of the table S.

\[
((S \text{ RENAME S\# AS SA})\{SA, CITY\} \text{ JOIN } (S \text{ RENAME S\# AS SB})\{SB, CITY\})
\text{ WHERE SA<SB}) \{SA, SB\}
\]

**Question 1:** Why Projection before Join?

**Answer:** The Join should happen only with CITY.

**Question 2:** Why the “SA<SB” condition?

**Answer:** Cut out the redundancy.
Below is another way to say it

```
(((S RENAME S# AS SA, SNAME AS NA, STATUS AS TA) JOIN
(S RENAME S# AS SB, SNAME AS NB, STATUS AS TB)
WHERE SA<SB) {SA, SB}
```

Its SQL form *might* look like the following:

```
Select SA, SB
From (S RENAME S# AS SA, SNAME AS NA,
    STATUS AS TA, CITY AS CA),
(S RENAME S# AS SB, SNAME AS NB,
    STATUS AS TB, CITY AS CB)
Where CA=CB AND SA<SB;
```

However, RENAME operation is applied on a table in SQL. Thus, we have the following

```
Select A.S# AS SA, B.S# AS SB
From S A, S B
Where A.CITY=B.CITY AND SA<SB;
```

**Question to Jon B.**: Is RENAME useful?

**Answer from Jon:**
6. Get supplier names for those who supply every part.

Its algebraic expression is as follows.

\[((\{S\} \text{ DIVIDEBY } \{P\} \text{ PER } \{SP\}) \text{ JOIN } S) \{\text{SNAME}\}\]

Although its conversion to an SQL query is not immediate, we notice that what we want is the SNAME for those rows in the table S such that for every part in the table P there exists a row in the table SP that connects the row in S and the row in P.

Thus, we want SNAME in S with the following restriction:

\[
\text{FOR ALL } P \text{ EXISTS } SP \text{ SP.S#}=S.S# \text{ AND } SP.P#=P.P#
\]
Thus, the corresponding SQL query could be the following:

```
Select SNAME
From S
Where FOR ALL
(
    Select *
    From P
    Where EXISTS
    (    
        Select *
        From SP
        Where SP.S#=S.S# AND SP.P#=P.P#
    )
);
```

Here FOR ALL is applied to each and every P row, while EXISTS is applied to any such row.
Unfortunately,

SQL does not support the FOR ALL facility. But, we can take a bypass via the following two facts (from finite math.?)

1. For any statement $P$, $\text{NOT NOT } P \equiv P$.

   For example, “It is not true that we don’t understand this stuff” means exactly “We understand this stuff”.

2. For any statement $P$, $\text{NOT FOR ALL } P \equiv \text{EXIST NOT } P$.

   For example, “It is not true that I know everybody” means exactly “There is somebody whom I do not know.”
Now what?

Hence, the above restriction, i.e.,

\[
\text{FOR ALL } P \text{ EXISTS } SP \ SP.S# = S.S# \text{ AND SP.P#} = P.P#
\]

can be converted to

\[
\text{NOT NOT (FOR ALL } P \text{ EXISTS } SP \ SP.S# = S.S# \text{ AND SP.P#} = P.P#)
\]

then, to

\[
\text{NOT EXISTS } P \text{ NOT EXISTS } SP \ SP.S# = S.S# \text{ AND SP.P#} = P.P#
\]

The last restriction says “such that there does not exist a P row for which there does not exist any SP row”, which certainly means “for all P row there exists a SP row...”.
Finally,...

we have the following SQL query for the ques-
tion:

```
Select SNAME
From S
Where NOT EXISTS
(
   Select *
   From P
   Where NOT EXISTS
   (  
      Select *
      From SP
      Where SP.S#=S.S# AND SP.P#=P.P#
   )
);
```

Here the first EXISTS is applied to each and
every P row, while the second EXISTS is applied
to any such row.
Homework

Carefully complete exercises 7.14–7.30 by going through the following steps.

1. Use the sample data as contained in the backcover to figure out the output.

2. Construct relational algebraic expression.

3. Apply the expression to the sample data to confirm you have done it right.

4. Construct as many SQL expression as possible, based on the algebraic one, type it in and compare your result with the two results you have got so far.
Something handy

We can also use WITH in a relational expression. For example, below shows another, but equivalent, expression for the last query.

```sql
WITH S {S#} AS T1,
SP WHERE P#=P#('P2') AS T2,
   T2{S#} AS T3,
T1 MINUS T3 AS T4,
T4 JOIN S AS T5,
T5 {SNAME} AS T6:
T6
```

In such a WITH statement, the expression preceding the ‘:’ requires no immediate evaluation. The system simply associates it with the stuff after the ‘:’, which, T6 in this case, denotes the final result. When the system sees this, it will retrieve its definition and go through with it.
Why this algebra?

Besides being used to define a scope for various data retrieval purpose, the main purpose of the set of the operations just defined is to allow the writing of relational expressions so that we can define a scope for data update (Chapter 5), integrity constraints (Chapter 8), derived rel-vars (Chapter 9), stability requirements (Chapter 15), security constraints (Chapter 16), etc..

In general, the expressions can be used as a high-level symbolic representation of the user's intent. Moreover, since it is at a high level, it is possible to apply various optimization rule to further simplify them. E.g.,

\[
((SP \ JOIN \ S) \ WHERE \ P# = P#('P2'))\{\text{SNAME}\}
\]

can be transformed into the following more efficient one

\[
((SP \ WHERE \ P# = P#('P2')) \ JOIN \ S) \ \{\text{SNAME}\}
\]
Additional operators

Although we only need the basic five: restrict, project, product, union, and difference, it is handy to have the other three, i.e., join, intersect, and divide. Moreover, numerous other operators are also suggested, such as SEMIJOIN, SEMIMUNUS, EXTEND, SUMMARIZE, and TCLOSE, etc..

Semijoin of A and B is just A JOIN B projected on the attributes of A. Thus, the body of such a join is the tuples of A that has counterpart in B.

In contrast, Semidifference is just the opposite, which is defined as A MINUS (A SEMIJOIN B).
What do the Parisian supply?

Given the following instance for S and the standard instance for SP,

<table>
<thead>
<tr>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
</tbody>
</table>

SP Semijoin S is the following:

<table>
<thead>
<tr>
<th>S#</th>
<th>P#</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>P1</td>
<td>300</td>
</tr>
<tr>
<td>S2</td>
<td>P2</td>
<td>400</td>
</tr>
<tr>
<td>S3</td>
<td>P2</td>
<td>200</td>
</tr>
</tbody>
</table>

Its SQL equivalent is the following:

```sql
select sp.*
from s_temp, sp
where s_temp.s#=sp.s#
```
Extension

The _Extend_ operator adds an additional attribute into a table. E.g.,

```sql
((EXTEND P ADD (WEIGHT*454) AS GMWT)
   WHERE GMWT>WEIGHT(5000.0))
```

After applying the above to the “standard instance” of P, we would have the following:

<table>
<thead>
<tr>
<th>P#</th>
<th>PNAME</th>
<th>COLOR</th>
<th>WEIGHT</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P6</td>
<td>Cog</td>
<td>Red</td>
<td>19.0</td>
<td>London</td>
</tr>
</tbody>
</table>

Its SQL equivalent is the following:

```sql
select P.*, P.WEIGHT*454 AS GMWT
from P
where P.WEIGHT*454>5000;
```
More about EXTEND

In general, the value of the expression EXTEND A ADD exp AS Z, is defined to be a relation with heading equal to the heading of A extended with the new attribute Z and with body consisting of all tuples t such that t is a tuple of A extended with a value for the new attribute Z that is counted by evaluating the expression exp on the value of A.

Thus, we can find out the shipment wright for a supplier as follows:

EXTEND (P JOIN SP) ADD (WEIGHT*QTY) AS SHIPWT

select SP.*, SP.QTY*P.Weight as SHWT
from sp, p
where sp.p#=p.p# and sp.s#='S1'
Aggregate operator

COUNT is just one example of the aggregate operators, which is to derive a single scalar value from the values appearing in some specified attribute of some specified relation. Typical examples are COUNT, SUM, AVG, MAX, MIN, ALL, AND ANY. For example,

\[
\text{SUM (SP WHERE S#=S#('S1'), QTY)}
\]

On the other hand, the following finds the total of all distinct shipment quantities for 'S1'.

\[
\text{SUM ((SP WHERE S#=S#('S1')) \{}QTY\})}
\]

An SQL example is as follows:

```
Select count(*), sum(QTY), max(QTY),
         min(QTY), avg(QTY)
from SP;
```
Grouping and ungrouping

We now discuss an additional operator dealing with groups.

\[ \text{SP GROUP (P#, QTY) AS PQ} \]

In SQL, we can’t do exactly the same thing:

\[ \text{select * from SP group by S#;} \]

but, we often do the following:

\[ \text{select count(*) from SP group by S#;} \]
Applying the algebraic expression to the SP table, we will get the following:

<table>
<thead>
<tr>
<th>S#</th>
<th>PQ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P#</td>
</tr>
<tr>
<td>S1</td>
<td>P1</td>
</tr>
<tr>
<td></td>
<td>P2</td>
</tr>
<tr>
<td></td>
<td>P3</td>
</tr>
<tr>
<td></td>
<td>P4</td>
</tr>
<tr>
<td></td>
<td>P5</td>
</tr>
<tr>
<td></td>
<td>P6</td>
</tr>
<tr>
<td>S2</td>
<td>P1</td>
</tr>
<tr>
<td></td>
<td>P2</td>
</tr>
<tr>
<td>S3</td>
<td>P2</td>
</tr>
<tr>
<td>S4</td>
<td>P2</td>
</tr>
<tr>
<td></td>
<td>P4</td>
</tr>
<tr>
<td></td>
<td>P5</td>
</tr>
</tbody>
</table>
What is going on?

This specific grouped table is defined as follows:

1. Its heading consists of the relation-valued attribute PQ, together with the other attributes of SP, i.e., S#.

2. Each tuple in the body of this new relation consists of the applicable S# value, s, together with a PQ value, pq, obtained as follows: 1) Each SP tuple is replaced with a tuple, x, in which the P# and QTY components are combined into a tuple valued component, y. 2) The y components of all such tuples x, in which the S# value is equal to s are “grouped” into pq.

It is not a surprise that SPQ UNGROUP PQ will get us back the original SP solution.
Summarize

If we apply the following,

SUMMARIZE SP PER SP {P#} ADD SUM(QTY) AS TOTQTY

We would get the following result:

<table>
<thead>
<tr>
<th>P#</th>
<th>TOTQTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>600</td>
</tr>
<tr>
<td>P2</td>
<td>1000</td>
</tr>
<tr>
<td>P3</td>
<td>400</td>
</tr>
<tr>
<td>P4</td>
<td>500</td>
</tr>
<tr>
<td>P5</td>
<td>500</td>
</tr>
<tr>
<td>P6</td>
<td>100</td>
</tr>
</tbody>
</table>

Conceptually, the relation SP is grouped into sets of tuples, one set for each P# value in P(P#), and then each group is used to generate one result tuple. Thus, in SQL,

```sql
select p#, sum(QTY) as totqty
from SP
group by p#
```
What does it do?

SUMMARIZE (P JOIN SP) PER P\{CITY\}
ADD COUNT AS NSP

It joins the two tables, then group the other pieces into group, each of which has the same City value. It then add counts the number of rows for each group. Hence, it essentially lists all the cities together with the number of parts stored there.

In SQL, we do the following:

```sql
select city, count(*)
from p, sp
where p.p#=sp.p#
group by city;
```
A couple of points

1. Multiple SUMMARIZEs is allowable. E.g.,

```
SUMMARIZE SP PER P{P#} ADD SUM(QTY) AS TOTQTY,
ADD AVG(QTY) AS AVGQTY
```

This is to find out the total and the average number of parts as supplied.

```
Select p#, sum(QTY), avg(QTY)
From sp
Group by p#;
```

2. If we want to find out the same information, but only for the larger shipments, for example, for those QTY exceeds 200, we can put on a restriction.

```
Select p#, sum(QTY), avg(QTY)
From sp
Where QTY>200
Group by p#;
```
3. Continuing with the first question. If we want the same information for part P1 only, we could use the Having clause.

Select p#, sum(QTY), avg(QTY)
From sp
Group by p#
Having p#='P2';

4. If we want to find out, for P2 only, the total and the average value of those quantity exceeding 200, we combine everything together.

Select p#, sum(QTY), avg(QTY)
From sp
Where QTY>200
Group by p#
Having p#='P2';
Group in SQL

When the system sees an SQL statement as follows:

```sql
Select P#, sum(QTY), avg(QTY)
From SP
Where QTY>200
Group by P#
Having P#='P2';
```

it chooses all the rows from the table SP such that the `Where` condition holds, then `Group` all the chosen rows by the same `P#`, then apply the `sum` and `avg` operations to those rows, finally, display the computed value only for those groups that meets the cut as specified by the `Having` clause.
Relation comparators

The syntax of relational comparison is as follows: `<relational expression> Θ <relational expression>`, where Θ can be any of the following: `=`, `≠`, `<=(subset of)`, `<(proper subset of)`, `≥(superset of)`, and `>(proper superset of)`. Below are some applications:

1. S `{CITY}`=P `{CITY}` (Is there a city where parts are stored, but no supplier is located there?)
2. S `{S# } > SP{S#}` (Is there a supplier that supplies nothing?)
3. S WHERE (((SP RENAME S# AS X) WHERE X=S#){P#}=P{P#}) (Who supplies all the parts?)

We use IS_EMPTY(<relation expression>) to test if a table is empty.
We can also use such operators as \textit{IN}, \textit{ANY} and \textit{ALL}. For example,

\begin{verbatim}
S WHERE (CITY IN ('Paris', 'London'))
\end{verbatim}

will pick up tuples that are located either in Paris, or London. In SQL,

\begin{verbatim}
select * from s
where city in ('Paris', 'London');
\end{verbatim}

\begin{verbatim}
S WHERE (STATUS < ANY (20, 40, 25))
\end{verbatim}

will pick up those suppliers whose status is less than \textit{any} of 20, 40, or 25. In SQL,

\begin{verbatim}
select * from s where status < ANY(20, 25);
\end{verbatim}

\textbf{Homework:} Try Exercises 7.32–7.50 (even numbered) by going through the same process as before.