Chapter 5
The Relational Model

The relational model can be regarded as a composition of three aspects, dealing with data structure, its integrity, and the manipulation, respectively. The most important structural terms are illustrated in the following figure. We discuss the structural part in this chapter, manipulative aspect in the following three chapters, and the integral part thereafter.
Types

Similar to any artificial language, any give type of RDB is either system-defined or user-defined. For example, INTEGER and CHAR are system defined, while, e.g., S# is user defined. Any type can be used to declare relational attributes.

Associated with a specific type, is the associated notion of the valid operators that can be legally applied to values of this type. E.g., for the type of INTEGER, the system provide operators “=” , “<”, and so on for the comparison purpose, it also provides all the arithmetic operations. On the other hand, string operations are not supported for the type of integers.
Values and variables

A value is essentially a constant, e.g., 3, or “DBA”, thus not associated with either time or space. It is just there. However, a value can be represented in the memory, and its representation is associated with time and space. For example, it takes 2 bytes to represent 3, represented as an integer, and it takes certain amount of time to retrieve it from the memory.

A value can thus exist in any locations, and by definition, a value itself can’t be updated.

In contrast, a variable is a holder of a representation of a value. It does occupy a space, and can certainly be updated. In particular, many variables can have the same value.

Homework: Exercise 5.2.
Everything has a type

A value is different from a variable. But, they do share one thing in common, both of them have a type. In fact, everything used in RDB will be declared of exactly one type, and before applying any operation, the system will check to see if the operands are of the “right” types for the operation.

For instance, it makes no sense to do the following

\[ P.WEIGHT + SP.QTY \]

and hence should be rejected, even though the fields involved both are of the numeric types.

**Homework:** Read the discussion related to polymorphism at the end of §5.2 and then complete Exercise 5.1.
Scalar and non-scalar types

A type is different from its representation. The former is a design issue, while the latter is an implementation issue, and should be hidden from the normal users.

A non-scalable type is a type whose values consisting of a set of user-visible, directly accessible components. An array is an example of such a type. On the other hand, a scalable type is just the opposite. It is also often referred to as an atomic type. A value is called scalable or non-scalable, depending on the category of its type.
Type definition

We will use the language *Tutorial D* to demonstrate some of the points of types. Below are examples to declare a user-defined type.

```plaintext
TYPE S#    POSSREP (CHAR);
TYPE NAME  POSSREP (CHAR);
TYPE P#    POSSREP (CHAR);
TYPE COLOR POSSREP (CHAR);
TYPE WEIGHT POSSREP (RATIONAL);
TYPE QTY    POSSREP (INTEGER);
```

Each of the above declaration will cause the system to make an entry in The catalog to describe that new type.

It is easy to drop a type.

DROP TYPE NAME
Possible representation

Let's consider a more complex example

TYPE POINT /*geometric points */
    POSSREP CARTESIAN (X RATIONAL, Y RATIONAL)
    POSSREP POLAR (R RATIONAL, THETA RATIONAL);

In general, every type has at least one possible representation, which automatically causes definition of the following operators: 1) A selector, with the same name as that of the type, allows the user to specify or select a value or that type. E.g., CARTESIAN(5.0, 2.5). 2) A set of THE_ operators for each of the component types, which allows the user to access the corresponding components. E.g., THE_X(P), which returns the x coordinate of point P.
Assume that the physical representation of points is in fact Cartesian coordinates. Then the system will provide certain highly protected operators that effectively access that physical representation, so that the type definer can implement the necessary CARTESIAN and POLAR selectors. For example,

OPERATOR CARTESIAN(X RATIONAL, Y RATIONAL) RETURNS (POINT);
BEGIN
VAR P POINT;
X component of P:=X;
Y component of P:=Y;
RETURN (p);
END
END OPERATOR;

OPERATOR POLAR(R RATIONAL, THETA RATIONAL) RETURNS (POINT);
RETURN(CARTESIAN (R*COS(THETA),
(R*SIN(THETA))));
END OPERATOR;
The type definer will then use those operators to implement the necessary THE_operators, e.g.,

```
OPERATOR THE_X(p POINT) RETURN(RATIONAL);
  RETURNS (X component of P);
END OPERATOR;
```

```
OPERATOR THEY_R(P POINT) RETURNS (RATIONAL);
  RETURN(SQRT(THE_X(P)**2+THE_Y(P)**2));
END OPERATOR;
```

For simpler types, we also can use the same operators. For example, QTY(100), QTY(((Q1-Q2)*2), and THE_QTY(Q).

**Questions:** Is it correct to say that the quantity of a shipment is 100? If not, what should we say about it?
Operator definition

It is easy to define user-defined operations. For example

OPERATOR ABS (Z RATIONAL) RETURNS (RATIONAL);
    RETURN (CASE
    WHEN Z >= 0.0 THEN =Z
    WHEN Z < 0.0 THEN -Z
    END CASE);
END OPERATOR;

OPERATOR REFLECT (P POINT) UPDATES(P);
    BEGIN;
    THE_X(P):=-THE_X(P);
    THE_Y(P):=-THE_Y(P);
    RETURN;
    END;
END OPERATOR;

Homework: Exercise 5.5, 5.6, and 5.7. Hint: The difference between 5.6 and 5.7 is that with 5.6, you only return a value, without changing it; and in 5.7, you actually change it.
Type conversion

Given the following type definition:

```
TYPE S# POSSREP (CHAR);
```

it is valid to invoke the following selector S# (‘S1’). Thus, loosely speaking, the above selector plays the role of a type conversion operation that converts character strings to supplier numbers. Other selectors play the similar roles.

Thus, the system will not regard the following as having a type error ... WHERE P#='p2', since it will realize that it can use the P# converter to convert the CHAR compared to type P# first, then do the comparison. Remember?

```
int i=5;
double d=i;
```
Hence, effectively, the comparison is rewritten as follows:

...WHERE P# = P#('P2')

This type of conversion is known as coercion, which could lead to program bugs. Thus, we conservatively require that all the comparands be of the same type; and the LHS and RHS of an assignment be of the same type, as well.

We certainly can use the "CAST" operator as well.

CAST_AS_RATIONAL (5)

Remember?

int j = (int) d;
Type generators

Some types can be generated using type generator. For example,

```
VAR SALES ARRAY INTEGER[12];
```

In this case, the expression `ARRAY INTEGER[12]` can be regarded as an invocation of the `ARRAY` type generator, which returns a specific type. Such a type is called a generated type, and can be used anywhere a type fits.

There are also associated selector and `THE_` operators for generated types. For example, `SALES[3]` can be used to get access to the third component of the associated array. We can also associate it with the normal assignment, and perhaps comparison, operators.

**Homework:** Exercises 5.12 and 5.14. **Hint:** For 5.12, just write out the possible representation for all the involved types in the tables.
Defining tuples

Given a collection of types, \( T_i, i \in [1, n] \), not necessarily distinct, a tuple on \( T_i \)'s, \( t \), is a set of ordered triples \(< A_i, T_i, v_i >\), where \( A_i \) is the attribute name of, \( T_i \) is the type of, and \( v_i \), of type \( T_i \), the value of, the \( i^{th} \) component of the tuple.

Moreover, we call \( n \) the degree, or arity, of \( t \); \(< A_i, T_i >\) an attribute of \( t \); \( v_i \) the attribute value of \( A_i \) of \( t \), and \( T_i \) the corresponding type for \( A_i \).

The complete set of the attributes of \( t \) is called its heading.
An example

Given the following tuple,

<table>
<thead>
<tr>
<th>Major_P#:P#</th>
<th>Minor_P#:P#</th>
<th>QTY:QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>P4</td>
<td>7</td>
</tr>
</tbody>
</table>

the attribute names are Major_P#, Minor_P#, and QTY. The corresponding types are P#, P# and QTY. Finally, the related values are P#('P2'), P#('P4'), and QTY(7). The degree of this tuple is 3.

In TUTORIAL D, we can use the following tuple generator to generate this tuple:

TUPLE{MAJOR_p# P#('P2'),MINOR_p#('P4'),QTY QTY(7)}
Properties of tuples

1. Every tuple contains exactly one value for each of its attributes.

2. Since a tuple is a set of components, there is no imposed order of those components.

3. Every subset of a tuple, including the empty subset, is again a tuple.

An empty tuple, denoted as TUPLE, plays a crucial role, as we will see later.
Operations on tuples

Two tuples, $t_1$ and $t_2$ are said to be equal to each other iff they have the same attributes and all the corresponding values are equal to each other.

Given the following tuple \( \text{ADDR} = \text{TUPLE}\{\text{STREET}'1600 Pennsylvania Ave.'}, \text{CITY}'Washington', \text{STATE}'DC', \text{ZIP}'20500'\}\), the tuple projection, \( \text{ADDR}\{\text{CITY, ZIP}\}\) returns \( \text{TUPLE}\{\text{CITY}'Washington', \text{ZIP}'20500'\}\), and the following extract operation \( \text{ZIP FROM ADDR}\) returns '20500'.
Relation values

The term *relation* can be defined as follows: given a collection of \( n \) types \( T_i \) \((i = 1, 2, \ldots, n)\), not necessarily all distinct, \( r \) is a *relation* on those types if it consists of two part, an *heading* and a *body*, where:

a. the heading is a set of \( n \) attributes of the form \( A_i : T_i \), where the \( A_i \)'s, which must be all distinct, are the attribute names of \( r \), and the \( T_i \)'s are the corresponding type names.

b. the body is a set of \( m \) tuples \( t \), where \( t \) in turn is a set of components of the form \( A_i : v_i \) in which \( v_i \) is a value of type \( T_i \), called the attribute value for attribute \( A_i \) of tuple \( t \).

The values \( m \) and \( n \) are called the *cardinality* and the *degree*, respectively, or relation \( r \).

**Homework:** Exercises 6.1, 6.2, and 6.3.
A couple of points

1. In terms of the tabular representation of a relation, the heading is the top row of column names and corresponding type names, the body is just the set of data rows.

2. Attribute $A_i$ is said to be of, or defined on, type $T_i$. Any number of distinct attributes can be of the same type.

3. Values of a given type do not need to always appear in a database.

4. A relation of degree one is said to be unary, a relation of degree $n$ is called $n$-ary.

5. A relation $r$ has heading $H$ is to say that $r$ is of type $\text{RELATION}(H)$. The latter is also used as the name of that type.
## An example

<table>
<thead>
<tr>
<th>S#:S#</th>
<th>SNAME:NAME</th>
<th>STATUS:STATUS</th>
<th>CITY: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>
</tbody>
</table>

The above table has four underlying types, and has two parts: a row of names, and a set of data rows. The row \((S1, Smith, 20, London)\) can be regarded as the following set of ordered pairs:

\[
\{S#: S#('s1'),
SNAME: NAME('Smith'),
STATUS: 20,
CITY: ‘London’\}

Thus, by definition, the above table is actually a *picture* of a relation. Strictly speaking, relation is what the definition says it is, and a table is a concrete instance of this relation at a moment.
Properties of relations

Relation possess certain properties, following the definition.

1. Each tuple contains exactly one value for each attribute. (?).
2. Tuples are unordered (?).
3. Attribute are unordered (?).
4. There are no duplicate tuples (?).

Also, by definition, a value could have any type, particularly, it could have a relation as its type. For example, in a relation S_SP could have an attribute PQ, telling, for each supplier, the parts it supplies and the corresponding quantities.

Finally, by the Closed World Assumption, if a tuple is not in the table, it is false.
Relation vs. tables

Although relation is sort of a formal name of a RDB table, there are some differences, including the following:

1. Each attribute in the heading of a relation has a type name, but it is dropped in the related table. Same thing happens to the component of each tuple in the table.

2. The types are simplified in the tables. For example, instead of P#('P2') we simply put down P2.

3. The columns in a table have to be ordered from left to right, while in a relation, the associated attributes constitute a set, thus having no order. Same things happens between the rows and the tuples in a table.

Homework: Read the subsection in pp. 151 and complete Exercises 6.9 and 6.18.
Operations on relations

Relations can be compared as sets in terms of equal, not equal, subset of, proper subset of, etc.. For example,

\[ S \{\text{CITY}\} = P \{\text{CITY}\} \]

or

\[ S \{\text{S}\#\} \text{ NOT} = SP \{\text{S}\#\} \]

We can also test if a tuple, \( t \), is a member of a relation, \( r \), \( (t \in r) \) and extract a tuple from a relation, TUPLE FROM \( r \).

There are numerous operations we can apply among relations, such as Restriction, projection, Product, etc., which constitute the manipulative aspect of RDB.
Relation variables

Below is the syntax defining a base relvar:

VAR <relvar name> BASE <relation type>
  <candidate key definition list>
  [<foreign key definition list>];

The <relation type> takes the form

RELATION {<attribute commalist>},

where each <attribute> is an ordered pair of the form

<attribute name> <type name>
An example

VAR P BASE RELATION{
  P#    P#
  PNAME NAME,
  COLOR COLOR,
  WEIGHT WEIGHT
  CITY   CHAR }
  PRIMARY KEY {P#};

VAR SP BASE RELATION{
  S#    S#
  P#    P#
  QTY   QTY}
  PRIMARY KEY {S#, P#}
  FOREIGN KEY {S#} REFERENCES S
  FOREIGN KEY {P#} REFERENCES P;

To drop an existing base relvar, use the following syntax

DROP VAR <relvar name>;

Homework: Exercises 6.4, and 6.10.
Update relvars

The syntax to update an existing relvar is the following:

<relvar name>:=<relational expression>;

For example, given another relvar R of the same type as the suppliers relvar S:

VAR R BASE RELATION
{S#,SNAME,STATEUS,CITY};

We can apply the following legal relational assignment:

R:=S;
R:=S WHERE CITY='London';
R:=S MINUS (S WHERE CITY='Paris');
Other operators

For convenience, Tutorial D also supports explicit INSERT, DELETE, and UPDATE operations. For example,

```
INSERT INTO S
RELATION {TUPLE {S# S#('S6'),
                 SNAME NAME ('Smith'),
                 STATUS 50,
                 CITY 'Rome'}};
```

The above is equivalent to the following:

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

**Homework:** Exercise 6.15.