1. SEMANTIC INTRODUCTION TO DATABASES

```
Database — From DATA and BASE (adj = low, mean, vile, etc). A place where data can be lost in a structured manner.
```

```
DBMS (Database Management System) — The software needed to set up highly complex inter-relational data structures, so that files can be lost in any convenient sequence (e.g. Index before data; First-in-last-out).
```

From a folklore dictionary.

This chapter defines fundamental concepts of databases. These concepts are described here in terms of the Semantic Binary Model (SBM) of data. A data model is a convention for the specification of the logical structure of real-world information. The cornerstone of the contemporary theory and technology of databases was the development of the Relational Data Model. The recent development of the new generation of data models — the semantic models — offers a simple, natural, implementation-independent, flexible, and nonredundant specification of information. The word semantic means that this convention closely captures the meaning of user’s information and provides a concise, high-level description of that information.

SBM is one of several existing semantic models. The various semantic models are roughly equivalent and have common principles, even though they somewhat differ in terminology and in the tools they use. SBM is simpler than most other semantic data models: it has a small set of sufficient tools by which all of the semantic descriptors of the other models can be constructed. After mastering SBM, a systems analyst may wish to explore more complex semantic models.

This chapter defines and discusses the concepts of a database, a database management system (DBMS), a database schema, modeling real-world information, categorization of real-world objects, relations between objects, graphic representation of database schemas, integrity constraints, quality of database schemas, subschemas, userviews, database languages, services of DBMS, and multimedia databases.

1.1. Databases, DBMS, Data Models

**General-purpose software system** — a software system that can serve a variety of needs of numerous dissimilar enterprises.
Example 1-1.
A compiler for a programming language.

**Application** — a software system serving the special needs of an enterprise or a group of similar enterprises.

Example 1-2.
The registration of students in a university.

**Application’s real world** — all the information owned by and subject to computerization in an enterprise or all such information which is relevant to a self-contained application within the enterprise.

Example 1-3.
The examples of this text constitute a case study. Its application world is the educational activities of a university. The information contains:

- A list of the university’s departments (including all the full and short names of each department)
- Personal data of all the students and their major and minor departments
- Personal data of all the instructors and their work information (including all the departments in which the instructor works and all the courses which the instructor teaches)
- The list of courses given in the university catalog
- The history of courses offered by instructors
- The history of student enrollment in courses and the final grades received

**Database** — an updatable storage of information of an application’s world and managing software that conceals from the user the physical aspects of information storage and information representation. The information stored in a database is accessible at a logical level without involving the physical concepts of implementation.

Example 1-4.
Neither a user nor a user program will try to seek the names of computer science instructors in track 13 of cylinder 5 of a disk or in "logical" record 225 of file XU17.NAMES.VERSION.12.84. Instead, the user will communicate with the database using some logical structure of the application’s information.

Rishe-DDS p. 1.2
Normally, a database should cover all the information of one application; there should not be two databases for one application.

**Database management system, DBMS** — a general-purpose software system which can manage databases for a very large class of the possible application worlds.

*Example 1-5.*

A DBMS is able to manage our university database and also completely different databases: an Internal Revenue Service database, an FBI *WANTED* database, a UN database on world geographical data, an Amtrak schedule, etc.

**Instantaneous database** — all the information represented in a database at a given instant. This includes the historic information which is still kept at that time.

The actual information stored in the database changes from day to day. Most changes are additions of information to the database.

*Example 1-6.*

A new student, a new instructor, new events of course offerings.

Fewer changes are deletions of information.

*Example 1-7.*

Historic information past the archival period; a course offering which was canceled before it was given.

Some changes are replacements: updates; correction of wrongly recorded information.

*Example 1-8.*

Update of the address of a student; correction of the student’s birth year (previously wrongly recorded).

Hence the life of a database can be seen as a sequence of instantaneous databases. The first one in the sequence is often the empty instantaneous database — it is the state before any information has been entered.

**Database model** — a convention of specifying the concepts of the real world in a form understandable by a DBMS. (Technically, it is an abstract data structure such that every possible instantaneous database of nearly every application’s world can be logically represented by an instance of that data structure.)

The following database models will be studied in this text:

- *Semantic Binary*, in which the information is represented by logical associations (relations) between pairs of objects and by the classification of

Rishe-DDS p. 1.3
objects into categories

- *Relational*, in which the information is represented by a collection of printable tables
- *Network*, in which the information is represented by a directed graph of records
- *Hierarchical*, in which the information is represented as a tree of records

The Semantic Binary Model is the most natural of the above models. It is the most convenient for specifying the logical structure of information and for defining the concepts of an application’s world. In this text, the other models will be derived from the Semantic Binary Model. The Relational, Network and Hierarchical models are dominant in today’s commercial market of database management systems.

### 1.2. Semantic Modeling

#### 1.2.1. Categorization of objects

**Object** — any item in the real world. It can be either a concrete object or an abstract object as follows.

```
Example 1-9.
Consider the application world of a university.
I am an object, if I am of interest to the university. My name is an object. The Information Systems Department and its name "Information Systems Department" are two distinct objects.
```

**Value**, or **concrete object** — a printable object, such as a number, a character string, or a date. A value can be roughly considered as representing itself in the computer or in any formal system.

```
Example 1-10.
My name and the name "Computer Science Department" are concrete objects. The grade 70 which has been given to a student in a course is also a concrete object.
```

**Abstract object** — a nonvalue object in the real world. An abstract object can be, for example, a tangible item (such as a person, a table, a country) or an event (such as an offering of a course by an instructor) or an idea (such as a course). Abstract objects cannot be represented directly in the computer.

This term is also used for a user-transparent representation of such an object in the Semantic Binary Model.
**Example 1-11.**
The Management Science Department, the student of the department whose name is Alex Johnson, and the course named "Chemistry" are three abstract objects.

**Category** — any concept of the application’s real world which is a unary property of objects. At every moment in time such a concept is descriptive of a set of objects which possess the property at that time.

Unlike the mathematical notion of a set, the category itself does not depend on its objects: the objects come and go while the meaning of the category is preserved in time. Conversely, a set does depend on its members: the meaning of a set changes with the ebb and flow of its members.

Categories are usually named by singular nouns.

**Example 1-12.**

*STUDENT* is a category of abstract objects. The set of all the students relevant to the application today is different from such a set tomorrow, since new students will arrive or will become relevant. However, the concept *STUDENT* will remain unaltered.

An object may belong to several categories at the same time.

**Example 1-13.**

One object may be known as a person and at the same time as an instructor and as a student.

**Example 1-14.**

Some of the categories in the world of our university are: *INSTRUCTOR, PERSON, COURSE, STUDENT, DEPARTMENT*.

**Disjoint categories** — Two categories are disjoint if no object may simultaneously be a member of both categories. This means that at every point in time the sets of objects corresponding to two disjoint categories have an empty intersection.

**Example 1-15.**

The categories *STUDENT* and *COURSE* are disjoint; so are *COURSE* and *DEPARTMENT* (even though there may be two different objects, a course and a department, both named "Physics").
The categories *INSTRUCTOR* and *STUDENT* are not disjoint (Figure 1-1); neither are *INSTRUCTOR* and *PERSON*.

**Figure 1-1.** Intersecting of categories.

**Subcategory** — A category is a *subcategory* of another category if at every point in time every object of the former category should also belong to the latter. This means that at every point in time the set of objects corresponding to a category contains the set of objects corresponding to any subcategory of the category.

**Example 1-16.**
The category *STUDENT* is a subcategory of the category *PERSON*. The category *INSTRUCTOR* is another subcategory of the category *PERSON* (Figure 1-2).

**Figure 1-2.** Subcategories.

**Abstract category** — a category whose objects are always abstract.

**Concrete category, category of values** — a category whose objects are always concrete.

**Example 1-17.**
*STUDENT* and *COURSE* are abstract categories. *STRING, NUMBER, and DIGIT* are concrete categories.

Many concrete categories, such as *NUMBER, STRING, and BOOLEAN*, have constant-in-time sets of objects. Thus, those concrete categories are actually indistinguishable from the corresponding sets of all numbers, all strings, and the Boolean values ({TRUE, FALSE}).

Rishe-DDS p. 1.6
Finite category — A category is finite if at no point in time an infinite set of objects may correspond to it in the application’s world.

Example 1-18.
The categories STUDENT, COURSE, and DIGIT are finite. The category NUMBER may be infinite.

Every abstract category is finite.

Example 1-19.
The database has a finite size. We cannot have an abstract category POINT containing information about every point in a plane.

1.2.2. Binary relations

Binary relation — any concept of the application’s real world which is a binary property of objects, that is, the meaning of a relationship or connection between two objects.

Example 1-20.
WORKS-IN is a relation relating instructors to departments. MAJOR-DEPARTMENT relates students to departments. NAME is a relation relating persons to strings. BIRTH-YEAR is a relation relating persons to numbers.

At every moment in time, the relation is descriptive of a set of pairs of objects which are related at that time. The meaning of the relation remains unaltered in time, while the sets of pairs of objects corresponding to the relation may differ from time to time, when some pairs of objects cease or begin to be connected by the relation.

Notation: xRy means that object x is related by the relation R to object y.

Example 1-21.
To indicate that an instructor i works in a department d, we write:

i WORKS-IN d

1.2.2.1. Types of binary relations: m:m, m:1, 1:m, 1:1

1. A binary relation R is many-to-one (m:1, functional) if at no point in time xRy and xRz where y ≠ z.
Example 1-22.

BIRTH-YEAR is an m:1 relation because every person has only one year of birth:

\[
\begin{array}{ccc}
\text{person}_1 & \text{BIRTH-YEAR} & 1970 \\
\text{person}_2 & \text{BIRTH-YEAR} & 1970 \\
\text{person}_3 & \text{BIRTH-YEAR} & 1969 \\
\text{person}_4 & \text{BIRTH-YEAR} & 1965 \\
\end{array}
\]

Example 1-23.

MAJOR-DEPARTMENT is also an m:1 relation, since every student has at most one major department, as in Figure 1-3.

Figure 1-3. A many-to-one relation.

2. A binary relation \( R \) is one-to-many (1:m) if at no point in time \( xRy \) and \( zRy \) where \( x \neq z \).

Example 1-24.

The relation MAJOR-DEPARTMENT is not 1:m, since a department may have many major students.

If, instead of the relation MAJOR-DEPARTMENT, we have the relation MAJOR-STUDENT between departments and students, then this relation would be 1:m, since every student can have at most one major department.

3. Relations which are of neither of the above types are called proper many-to-many (m:m).

Rishe-DDS p. 1.8
Example 1-25.

WORKS-IN is a proper m:m relation because every instructor can work in many departments and every department may employ many instructors, as in Figure 1-4.

Figure 1-4. A m:m relation

4. A binary relation which is both m:1 and 1:m (always) is called one-to-one (1:1).

Example 1-26.

If courses are identified by their names, then the relation COURSE-NAME is 1:1, meaning that every course has at most one name, and no character string is the name of two different courses, as in Figure 1-5.

Figure 1-5. A 1:1 relation:

\[ course_i, \text{COURSE-NAME}, \text{name}_i \]
Example 1-27.

Suppose that in the current situation in our real world, the following is true:
(a) Every registered person has at most one name, and no two persons have the same name.

This does not mean that NAME is a 1:1 relation between persons and strings. NAME would be a 1:1 relation if condition a were true at all times: past, present, and future.

The following diagram shows the classification of all relations:

![Classification of relations](image)

**Figure 1-6.** Classification of relations.

5. A binary relation is **proper m:1** if it is m:1 and not 1:1.
6. A binary relation is **proper 1:m** if it is 1:m and not 1:1.

Example 1-28.

All of the types of relations mentioned in the previous example are proper.

Since the COURSE-NAME is 1:1, it is also 1:m, m:1, and m:m. Since this relation is proper 1:1, it cannot be proper 1:m, proper m:1, or proper m:m.

1.2.2.2. Categories as domains and ranges of relations

**Domain** and **range** of a binary relation:

**Domain of relation R** — a category C that satisfies the following two conditions:

a. Whenever xRy, then x belongs to C (at every point in time for every pair of objects)

b. No proper subcategory of C satisfies condition a

Rishe-DDS p. 1.10
Range of relation $R$ — a category $C$ that satisfies:

a. Whenever $xRy$, then $y$ belongs to $C$ (at every point in time for every pair of objects)

b. No proper subcategory of $C$ satisfies $a$

Example 1-29.
The domain of COURSE-NAME is the category COURSE and its range is the category STRING. The domain of WORKS-IN is INSTRUCTOR and the range is DEPARTMENT.

Total binary relation — A relation $R$ whose domain is $C$ is total if at all times for every object $x$ in $C$ there exists an object $y$ such that $xRy$. (At different times different objects $y$ may be related to a given object $x$.)

Note: No relation needs to be total on its domain.

Example 1-30.
Although the domain of the relation BIRTH-DATE is the category PERSON, the date of birth of some relevant persons is irrelevant or unknown. Thus, the relation BIRTH-DATE is not total.

1.2.2.3. Attributes

Some binary relations are often called attributes.

Attribute — A functional relation (i.e., m:1 or 1:1) whose range is a concrete category.

Example 1-31.

- first-name — attribute of PERSON, range: String (m:1)
- birth-year — attribute of PERSON, range: 1880..1991 (m:1)

The phrase ‘‘$a$ is an attribute of $C$’’ means: $a$ is an attribute, and its domain is the category $C$.

Example 1-32.
Last-name, first-name, and birth-year are attributes of PERSON.
1.2.3. Nonbinary relationships

Nonbinary relationships — real-world relationships that bind more than two objects in different roles.

Example 1-33.
There is a relationship between an instructor, a course, and a quarter in which the instructor offers the course.

Such complex relationships are regarded in the Semantic Binary Model as groups of several simple relationships.

Example 1-34.
The nonbinary relationship of the previous example is represented in the Semantic Binary Model by a fourth object, an offering, and three binary relations between the offering and the instructor, the quarter, and the course, as in Figure 1-7.

![Figure 1-7. A trinary relationship.](image)

In general, the Semantic Binary Model represents any nonbinary relation as:

\( a. \) An abstract category of events. Each event symbolizes the existence of a relationship between a group of objects.

\( b. \) Functional binary relations, whose domain is category (a). Each of those functional binary relations corresponds to a role played by some objects in the nonbinary relation.

Thus, the fact that objects \( x_1, \ldots, x_n \) participate in an \( n \)-ary relation \( R \) in roles \( R_1, \ldots, R_n \) is represented by:

\( a. \) An object \( e \) in the category \( R' \)

\( b. \) Binary relationships \( eR_1x_1, \ldots, eR_nx_n \)
Example 1-35.
The information about a course offered by an instructor during a quarter could be considered a ternary relation between instructors, courses, and quarters. In the Semantic Binary Model, we solve this problem by representing this information as a category \textit{COURSE-OFFERING} and three functional relations from \textit{COURSE-OFFERING}: \textit{THE-INSTRUCTOR}, \textit{THE-COURSE}, and \textit{THE-QUARTER}.

Instructor \(i\) has offered course \(c\) in quarter \(q\) if and only if there exists a course-offering \(o\), such that:

\[
\begin{align*}
o & \text{ THE-INSTRUCTOR } i \\
o & \text{ THE-COURSE } c \\
o & \text{ THE-QUARTER } q
\end{align*}
\]

1.2.4. Instantaneous databases

**Formal representation of an instantaneous binary database** — as a set of \textit{facts}, unary and binary:

**Unary fact** — a statement that a certain abstract object belongs to a certain category.

\textit{Example 1-36}.
(The person whose name is "Jane Howards") is a \textit{student}.

**Binary fact** — a statement that there is a certain relationship between two given objects.

\textit{Example 1-37}.
The \textit{birth-year} of (the person whose name is "Jane Howards") is 1968.

\textit{Example 1-38}.
(The instructor whose name is "John Smith") \textit{works in} (the department whose name is "Information Systems")

\textit{Example 1-39}.
The \textit{final grade} of (the enrollment of (the student whose name is "Jack Brown") in (the offering of (the course named "Basic Chemistry") by (the instructor named "Veronica

Risse-DDS p. 1.13
is 100.

Although this fact relates to the past, it is still relevant and thus is a part of today’s instantaneous database.

Note: In order to be in the current instantaneous database, the fact must have been explicitly or implicitly entered at some time and never canceled since.

### 1.2.5. Semantic binary schemas

A **semantic binary schema** is a description of the names and the properties of all the categories and the binary relations existing in an application’s world.

All the instantaneous databases under the schema should have only those categories and relations listed in the schema. The sets of pairs of objects corresponding, in the instantaneous database, to the categories, and the sets of pairs of objects corresponding to the relations, should satisfy the properties indicated in the schema.

The schema should list the following properties of the categories and relations: the subcategories, the domains and ranges of the relations, and the types of the relations (*proper m:m, proper m:1, proper 1:m, 1:1*).

**Semantic Binary Model** is a convention of specification of the structure of the real-world information in the form of a semantic binary schema. Technically, the model can be regarded as the set of all the possible binary instantaneous databases.

### 1.2.6. Schema diagrams

This section shows how schemas can be represented graphically in two dimensions.

1. In a schema diagram, categories are shown by rectangles.

#### Example 1-40.

- **STUDENT** — category

2. Relations from abstract categories to concrete categories are shown inside the boxes of the domain-categories as follows:
relation: range type

The range is specified as a programming language data-type. (We will use the style of Pascal here.)

Example 1-41.

DEPARTMENT

name: String 1:m

Usually, relations between abstract and concrete categories are m:1. This is the default type of relations whose ranges are concrete categories, and it need not be explicitly specified in the schema for such relations.

Example 1-42.

- PERSON –category
- last-name –attribute of PERSON, range: String (m:1)
- first-name –attribute of PERSON, range: String (m:1)
- birth-year –attribute of PERSON, range: 1870..1990 (m:1)
- address –attribute of PERSON, range: String (m:1)

PERSON

last-name: String
first-name: String
birth-year: 1870..1990
address: String

3. Relations between abstract categories are shown by arrows between the categories’ rectangles. (The direction of the arrow is from the domain to the range.) The name and type of the relation are indicated on the arrow. The default for the type of relations between abstract categories is m:m.

Example 1-43.

Figure 1-8 represents:

- DEPARTMENT –category

Rische-DDS p. 1.15
4. Subcategories’ rectangles are connected to their supercategories’ rectangles by arrows with dashes.

Example 1-44.
Figure 1-9 represents:

- PERSON —category
- STUDENT —subcategory of PERSON

Figure 1-9. Subcategories.

5. The disjointness of categories is indicated implicitly:
   
   a. Two categories which have a subcategory in common are not disjoint. (The common subcategory does not have to be their immediate subcategory, that is, it may be a subcategory of a subcategory and so on.)

Example 1-45.
No two of the categories in Figure 1-10 are disjoint from each other.
b. Two categories which are subcategories of one category (not necessarily immediate subcategories) are considered *not* disjoint, unless otherwise declared in an appendix to the schema.

*Example 1-46.*

No two of the categories in Figure 1-11 are disjoint from each other.

\[ \text{Example 1-47.} \]

\[ \text{STUDENT} \quad \text{COURSE} \]

Rishe-DDS p. 1.17
Example 1-48.
A semantic schema for a university application is given at the end of this book. This will be the principal reference schema used in the examples throughout the book.

1.2.7. Abstraction of a database storage structure

The physical implementation of a database is a responsibility of the DBMS. The implementation of the database should be completely transparent to the database users, including the database designers and systems analysts.

Nevertheless, it may be helpful to the reader to have a general idea of how the database may be implemented. This section shows a possible idea of database implementation. This is a simplistic implementation. Efficiency is not a concern here. It would be, of course, a concern in any actual implementation.

We can represent the abstract objects as integers. The DBMS will enumerate all the abstract objects. The numbers assigned to the objects will be invisible to the user, as will all the implementational details.

We can represent the instantaneous database as a large table. Every row contains a fact: a unary fact (object — category), or a binary fact (object — relation — object). This table can be implemented as a file.

Example 1-49.
The following is a fragment of a file representing an instantaneous database. The fragment consists of one unary fact and three binary facts.

<table>
<thead>
<tr>
<th>object#</th>
<th>relation or category</th>
<th>object# or value</th>
</tr>
</thead>
<tbody>
<tr>
<td>o21</td>
<td>COURSE-ENROLLMENT</td>
<td>o18</td>
</tr>
<tr>
<td>o21</td>
<td>THE-STUDENT</td>
<td>o17</td>
</tr>
<tr>
<td>o21</td>
<td>THE-OFFERING</td>
<td></td>
</tr>
<tr>
<td>o21</td>
<td>FINAL-GRADE</td>
<td>100</td>
</tr>
</tbody>
</table>

Example 1-50.
The following figure shows an implementation of an instantaneous database for the university application. It specifies the instantaneous database which will be used in examples throughout this book.
<table>
<thead>
<tr>
<th>category</th>
<th>object#</th>
<th>relation</th>
<th>object# or value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT</td>
<td>o1</td>
<td>NAME</td>
<td>Computer Science</td>
</tr>
<tr>
<td></td>
<td>o1</td>
<td>NAME</td>
<td>CS</td>
</tr>
<tr>
<td></td>
<td>o2</td>
<td>NAME</td>
<td>Mathematics</td>
</tr>
<tr>
<td></td>
<td>o2</td>
<td>NAME</td>
<td>Math</td>
</tr>
<tr>
<td></td>
<td>o3</td>
<td>NAME</td>
<td>Physics</td>
</tr>
<tr>
<td></td>
<td>o4</td>
<td>NAME</td>
<td>Arts</td>
</tr>
<tr>
<td></td>
<td>o5</td>
<td>NAME</td>
<td>Economics</td>
</tr>
<tr>
<td>COURSE</td>
<td>o6</td>
<td>NAME</td>
<td>Databases</td>
</tr>
<tr>
<td></td>
<td>o7</td>
<td>NAME</td>
<td>Gastronomy</td>
</tr>
<tr>
<td></td>
<td>o8</td>
<td>NAME</td>
<td>Football</td>
</tr>
<tr>
<td>QUARTER</td>
<td>o9</td>
<td>YEAR</td>
<td>1990</td>
</tr>
<tr>
<td></td>
<td>o9</td>
<td>SEASON</td>
<td>Fall</td>
</tr>
<tr>
<td></td>
<td>o10</td>
<td>YEAR</td>
<td>1990</td>
</tr>
<tr>
<td></td>
<td>o10</td>
<td>SEASON</td>
<td>Winter</td>
</tr>
<tr>
<td></td>
<td>o11</td>
<td>YEAR</td>
<td>1990</td>
</tr>
<tr>
<td></td>
<td>o11</td>
<td>SEASON</td>
<td>Spring</td>
</tr>
<tr>
<td>INSTRUCTOR</td>
<td>o12</td>
<td>LAST-NAME</td>
<td>Brown</td>
</tr>
<tr>
<td></td>
<td>o12</td>
<td>FIRST-NAME</td>
<td>George</td>
</tr>
<tr>
<td></td>
<td>o12</td>
<td>BIRTH-YEAR</td>
<td>1956</td>
</tr>
<tr>
<td></td>
<td>o12</td>
<td>ADDRESS</td>
<td>112 Lucky Dr.</td>
</tr>
<tr>
<td></td>
<td>o12</td>
<td>WORKS-IN</td>
<td>o1</td>
</tr>
<tr>
<td></td>
<td>o12</td>
<td>WORKS-IN</td>
<td>o2</td>
</tr>
<tr>
<td></td>
<td>o13</td>
<td>LAST-NAME</td>
<td>Watson</td>
</tr>
<tr>
<td></td>
<td>o13</td>
<td>FIRST-NAME</td>
<td>Mary</td>
</tr>
<tr>
<td></td>
<td>o13</td>
<td>BIRTH-YEAR</td>
<td>1953</td>
</tr>
<tr>
<td></td>
<td>o13</td>
<td>ADDRESS</td>
<td>231 Fortune Dr.</td>
</tr>
<tr>
<td></td>
<td>o13</td>
<td>WORKS-IN</td>
<td>o3</td>
</tr>
<tr>
<td></td>
<td>o14</td>
<td>LAST-NAME</td>
<td>Blue</td>
</tr>
<tr>
<td></td>
<td>o14</td>
<td>FIRST-NAME</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td>o14</td>
<td>BIRTH-YEAR</td>
<td>1950</td>
</tr>
<tr>
<td></td>
<td>o14</td>
<td>ADDRESS</td>
<td>536 Orange Dr.</td>
</tr>
<tr>
<td></td>
<td>o14</td>
<td>WORKS-IN</td>
<td>o2</td>
</tr>
</tbody>
</table>

**Figure 1-12.** An instantaneous semantic database for the university application.
<table>
<thead>
<tr>
<th>category</th>
<th>object#</th>
<th>relation</th>
<th>object# or value</th>
</tr>
</thead>
<tbody>
<tr>
<td>COURSE-OFFERING</td>
<td>o15</td>
<td>THE-INSTRUCTOR</td>
<td>o12</td>
</tr>
<tr>
<td></td>
<td>o15</td>
<td>THE-COURSE</td>
<td>o6</td>
</tr>
<tr>
<td></td>
<td>o15</td>
<td>THE-QUARTER</td>
<td>o9</td>
</tr>
<tr>
<td></td>
<td>o16</td>
<td>THE-INSTRUCTOR</td>
<td>o12</td>
</tr>
<tr>
<td></td>
<td>o16</td>
<td>THE-COURSE</td>
<td>o7</td>
</tr>
<tr>
<td></td>
<td>o16</td>
<td>THE-QUARTER</td>
<td>o9</td>
</tr>
<tr>
<td></td>
<td>o17</td>
<td>THE-INSTRUCTOR</td>
<td>o12</td>
</tr>
<tr>
<td></td>
<td>o17</td>
<td>THE-COURSE</td>
<td>o8</td>
</tr>
<tr>
<td></td>
<td>o17</td>
<td>THE-QUARTER</td>
<td>o9</td>
</tr>
<tr>
<td>STUDENT</td>
<td>o18</td>
<td>LAST-NAME</td>
<td>Victory</td>
</tr>
<tr>
<td></td>
<td>o18</td>
<td>FIRST-NAME</td>
<td>Elizabeth</td>
</tr>
<tr>
<td></td>
<td>o18</td>
<td>BIRTH-YEAR</td>
<td>1966</td>
</tr>
<tr>
<td></td>
<td>o18</td>
<td>ADDRESS</td>
<td>100 Sun St.</td>
</tr>
<tr>
<td></td>
<td>o18</td>
<td>MAJOR</td>
<td>o1</td>
</tr>
<tr>
<td></td>
<td>o18</td>
<td>MINOR</td>
<td>o5</td>
</tr>
<tr>
<td></td>
<td>o19</td>
<td>LAST-NAME</td>
<td>Howards</td>
</tr>
<tr>
<td></td>
<td>o19</td>
<td>FIRST-NAME</td>
<td>Jane</td>
</tr>
<tr>
<td></td>
<td>o19</td>
<td>BIRTH-YEAR</td>
<td>1965</td>
</tr>
<tr>
<td></td>
<td>o19</td>
<td>ADDRESS</td>
<td>200 Dorms</td>
</tr>
<tr>
<td></td>
<td>o19</td>
<td>MAJOR</td>
<td>o4</td>
</tr>
<tr>
<td></td>
<td>o19</td>
<td>MINOR</td>
<td>o5</td>
</tr>
<tr>
<td></td>
<td>o20</td>
<td>LAST-NAME</td>
<td>Wood</td>
</tr>
<tr>
<td></td>
<td>o20</td>
<td>FIRST-NAME</td>
<td>Michael</td>
</tr>
<tr>
<td></td>
<td>o20</td>
<td>BIRTH-YEAR</td>
<td>1964</td>
</tr>
<tr>
<td></td>
<td>o20</td>
<td>ADDRESS</td>
<td>110 Dorms</td>
</tr>
<tr>
<td></td>
<td>o20</td>
<td>MAJOR</td>
<td>o4</td>
</tr>
<tr>
<td></td>
<td>o20</td>
<td>MINOR</td>
<td>o5</td>
</tr>
<tr>
<td>COURSE-ENROLLMENT</td>
<td>o21</td>
<td>THE-STUDENT</td>
<td>o18</td>
</tr>
<tr>
<td></td>
<td>o21</td>
<td>THE-OFFERING</td>
<td>o17</td>
</tr>
<tr>
<td></td>
<td>o21</td>
<td>FINAL-GRADE</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>o22</td>
<td>THE-STUDENT</td>
<td>o19</td>
</tr>
<tr>
<td></td>
<td>o22</td>
<td>THE-OFFERING</td>
<td>o17</td>
</tr>
<tr>
<td></td>
<td>o22</td>
<td>FINAL-GRADE</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>o23</td>
<td>THE-STUDENT</td>
<td>o19</td>
</tr>
<tr>
<td></td>
<td>o23</td>
<td>THE-OFFERING</td>
<td>o15</td>
</tr>
<tr>
<td></td>
<td>o23</td>
<td>FINAL-GRADE</td>
<td>80</td>
</tr>
</tbody>
</table>

**Figure 1-12. Continued.**
1.3. Integrity Constraints

**Integrity constraints** (synonyms: integrity law, integrity rules) — rules attached to a database in order to detect obvious user errors when updating the database.

1. **Static integrity constraints** — rules to detect instantaneous databases which cannot correspond to any probable state of the application’s world in the past, present, or future, regardless of the database’s update history.

   *Example 1-51.*
   
   These are some static integrity constraints in our university:
   
   - No one has two last names
   - Every student has at most one major department
   - First names of people are composed only of letters
   - Students may not participate in a course before they were born or receive a grade in a course before they are 15 years old.

   A static integrity law can be regarded as a Boolean function from the set of all the instantaneous databases which are well-formed according to the schema or the database model. This function assigns the value *false* to those instantaneous databases which cannot correspond to any probable state of the application’s world.

2. **Dynamic integrity constraints** — as above, but the domain of the function is the set of transitions between instantaneous databases, and *false* is assigned to highly improbable transitions between states of the application’s world.

   *Example 1-52.*
   
   The following is a dynamic integrity constraint:
   
   The catalog of courses is unerasable — a course, once entered, may not be removed.

   *Example 1-53.*
   
   If we wish to record the sex of persons in our database and we are sure that nobody’s sex is ever recorded wrongly (this is usually a dangerous and unreasonable assumption), and we further assume that a woman cannot become a man, then the following would be another dynamic constraint:
   
   Once the sex of $x$ is *female* in an instantaneous database, the sex of $x$ remains *female* in the next instantaneous database.
Note:

(i) Very often some of the integrity constraints are captured by the schema.

(ii) Integrity constraints should be distinguished from *implementational restrictions*:

**Implementational restrictions** — the inability of the database to represent some possible situations of the application’s world or the inability to represent them in a logical, natural, nonredundant, error-avoiding, flexible way.

Implementational restrictions are caused by considerations such as hardware, software, database model, effort, time, and expenses.

---

**Example 1-54.**

If for the application world of the university we use a database model less powerful than the Semantic Binary Model, then our implementational considerations may require that every instructor is uniquely identified by social security number. This is not the case in the real world of the university, because sometimes an instructor receives a social security number only several months after being hired and becoming of relevance to the university database.

The aforementioned is a static implementational restriction. To cope with it we have either to delay the recording of new instructors or to supply them with some temporary numbers.

If our implementation further requires that the social security number of a person should remain constant in time, then this would constitute a dynamic restriction. Supplying temporary numbers would not help in this case. Also, this dynamic restriction would not allow for correction of a wrongly recorded social security number (due to a data-entry clerk’s mistake). In practice, such a correction may be possible but with an extremely high cost in terms of the programming effort and with a chance to inadvertently corrupt the database.

---

**1.4. Schema Quality Criteria**

We have defined the term *schema* for the Semantic Binary Model. The following is a more general definition of the term, regardless of the database model.

*Schema* — a description, in terms of a database model, of the concepts and the information structure of an application’s world. It may be the actual data structure of a database. A schema describes all the possible instantaneous databases for one given application’s world.

---

**Example 1-55.**

A schema for our university application should outline the basic relevant concepts of the university, such as *student, instructor*, etc., and the kinds of information to be gathered about them. The schema will not allow the database to contain information about salaries of the instructors or about girl- or...
boyfriends of students since these are outside the scope of the application’s world.

A schema is called **high quality** if it satisfies the following criteria:

1. **Natural**

   The schema describes the concepts of its application’s world naturally:
   - The schema describes the objects, categories, and relations as they are in the real world.
   - The users can translate ideas easily in both directions between the concepts of the schema and the natural concepts of the application world.

2. **Nonredundant**

   The schema contains very little or no redundancy. **Redundancy** is the possibility of representing a fact of the application’s world more than once in the same instantaneous database (so that if one of the representations is removed from the database, no information is lost; that is, all of the information represented by the instantaneous database remains unaltered).

   The redundancy should be avoided *not* in order to improve the storage efficiency — the storage is not that expensive nowadays. Moreover, the redundancy in the schema is not directly related to the redundancy in the physical storage: a logically nonredundant schema may be physically implemented by a redundant physical structure in order to improve the access-time efficiency.

   The redundancy should be avoided primarily in order to prevent inconsistency of the database and its update anomalies. When two facts in the database represent the same information, and that information is updated, the user may forget to update both facts. In this case, after the update the two facts would contradict. This contradiction may cause unpredictable behavior of many application programs. The ramifications would be much worse than the local incorrectness of a fact in the database.

   When the redundancy is needed for the convenience of the users, it should be introduced into the userviews (to be defined in the next section) but not into the schema.

   **Example 1-56.**

   The following is a fragment of a redundant *wrong* schema:

   ```
   COURSE
   ENROLLMENT
   final-grade: 0..100
   student’s-address: String
   ```

   Suppose a student $s$, whose address is $a$, has two enrollments, $e_1$ and $e_2$. Then,
the following facts (among others) are logically recorded in the database:

a. \( s \) ADDRESS \( a \)

b. \( e_1 \) THE-STUDENT \( s \)

c. \( e_2 \) THE-STUDENT \( s \)

d. \( e_1 \) STUDENT’S-ADDRESS \( a \)

e. \( e_2 \) STUDENT’S-ADDRESS \( a \)

The facts (d) and (e) can be inferred from the facts (a) through (c). Thus, (d) and (e) can be omitted from the database without altering the information represented by the database. These facts are redundant; the schema should not have allowed their entrance in the first place.

If the following relation were omitted from the above wrong schema,

\[
\text{address} \quad \text{– attribute of } \quad \text{PERSON, range: String } \quad (m:1)
\]

then the schema would still remain redundant and wrong, because fact (e) can be inferred from fact (d). (Additionally, it would be a problem to record the address of a student who has no enrollments yet).

In some database models we cannot eliminate the redundancy completely. When we have to have some redundancy, we should at least bind it by integrity constraints. When such constraints are implemented, the user is forced to update all the related facts simultaneously.

**Example 1-57.**

If we cannot avoid having the redundant schema of the previous example, we can at least try to enforce the following integrity constraint:

Whenever

\[
\begin{align*}
\text{address} & \quad \text{– attribute of } \quad \text{PERSON, range: String } \quad (m:1) \\
\text{THE-STUDENT} & \quad \text{– attribute of } \quad \text{PERSON, range: Student } \quad (m:1) \\
\text{STUDENT’S-ADDRESS} & \quad \text{– attribute of } \quad \text{PERSON, range: Student } \quad (m:1)
\end{align*}
\]

then

\[
\text{STUDENT’S-ADDRESS} \quad \text{– attribute of } \quad \text{PERSON, range: Student } \quad (m:1)
\]

3. **Nonrestrictive**

The schema does *not impose implementational restrictions*; that is, every situation probable in the real world of the application is fully representable under the schema.

**Example 1-58.**

A schema containing the following relation would prevent very senior citizens from entering our university:

\[
\text{birth-year} \quad \text{– attribute of } \quad \text{PERSON, range: 1900..2000 } \quad (m:1)
\]
This schema imposes an avoidable implementational restriction, and thus it is *wrong*.

4. **Covering integrity constraints**

The schema covers by itself as many integrity constraints as possible; that is, the class of instantaneous databases formally possible according to the schema is not much larger than the class of all possible situations of the real world.

Constraints that are *not* expressed in the schema cause these problems:

- They are hard to formulate and to specify.
- They are seldom enforced by the DBMS. Thus, they require a substantial application programming effort for their enforcement, are often implemented incorrectly, and usually prevent direct interaction between the user updating the database and the DBMS (the user may not use the standard language for simple updates, which is supplied by most DBMS).
- The users and application programmers often forget or misunderstand such constraints.

![Figure 1-13. A good schema.](image)

**Example 1-59.**

Figure 1-14 shows a fragment of a *poor* schema, with respect to the coverage of the integrity constraints by the schema.

It requires an integrity constraint not expressed in the schema:

> For no instructor are there two events of his or her work in the same department.

A better schema fragment is in Figure 1-13.

Rishe-DDS p. 1.25
5. **Flexible**

The schema is *flexible*: if probable changes in the *concepts* of the application world occur, the schema would not have to undergo drastic changes.

6. **Conceptually minimal**

The schema is *conceptually minimal*: it does not involve concepts which are irrelevant in the application’s real world and limits the accumulation of information which is irrelevant in that world.

*Example 1-60.*

The following would be irrelevant and **wrong** in the schema of our university:

- BEAUTIFUL — subcategory of STUDENT

*Example 1-61.*

The following would be irrelevant and **wrong** in the schema of our university, unless it is unavoidable due to technical problems:

- enrollment-number — attribute of ENROLLMENT, range: Integer

The most important issue of the database design is the design of a high-quality schema within the restrictions of the available DBMS and database model. A low-quality schema increases the chances of corruption of the data, makes it very hard to use and maintain the database, and makes it very hard, if not impossible, to adjust the database to the changing concepts of the

Rishe-DDS p. 1.26
application’s real world.

It is easy to design a high-quality schema in the Semantic Binary Model. The task is much harder in most other models. Moreover, it is usually impossible to describe an application world by a schema in the Relational, Network, or Hierarchical model with the same high quality as with which that application can be described in the Semantic Binary Model.

The following chapters will introduce methodologies to design conceptually adequate schemas in the Relational, Network, and Hierarchical models. Those schemas will be close to the highest quality possible within the restrictions of the respective models. In those methodologies, a semantic binary schema is designed first, and then the schema is translated into the model supported by the DBMS which will service the application.

1.5. Subschemas and Userviews

Subschema — a part of the schema, provided this part in itself can constitute a schema for some application world.

Example 1-62.
The following figure shows a schema for a very small application world. The only relevant information in that world is the names of the courses.

```
COURSE
name: String
```

This schema is a subschema of the University Binary Schema of Figure ___.

A subschema of a binary schema can be obtained by removing some of the categories and some of the relations (provided, whenever a category is removed, every relation whose domain or range is the category is also removed).

The primary use of the subschemas is to provide subpopulations of the database users with a partial view of the database information. The user of a subschema may regard it as if it were the entire schema and need not be aware of the existence of the information beyond the subschema. The DBMS will conceal from such a user all the information beyond the subschema. This brings the following benefits:

- Users do not have to understand the information concepts which are irrelevant to their activities.
- Users are prevented from accidentally corrupting the information which they had no business to access in the first place but accessed in error instead of the relevant information.

Rishe-DDS p. 1.27
• Malicious users are prevented from accessing information beyond that which they are entitled to access.

• When the database is extended by adding new concepts to the schema, and those new concepts do not affect some existing programs using subschemas, those programs need not be modified.

Example 1-63.
The subschema of Figure 1-15 covers the names of the courses and the seasons in which the courses are offered.

![Diagram of subschema for seasons of courses](image)

Figure 1-15. Subschema for seasons of courses.

Normally, the schema is partitioned into nondisjoint subschemas according to the needs of the different divisions within the enterprise and the different subapplications.
Figure 1-16. The users access the database through subschemas.

Figure 1-17. The schema is partitioned into subschemas, which need not be disjoint.

Example 1-64.
Figure 1-18 shows the subschema used by the Personnel Office.

A generalization of the subschema concept, the userview, is defined in the remainder of this section.

Rishe-DDS p. 1.29
Inference rules — rules by which new information can be deduced from the information that the users have entered into the database.

Example 1-65.

The following are some parts of the information recorded in the university database:

- For every instructor, the classes he or she teaches
- For every student, the classes he or she takes

An inference rule:

- If $s$ takes a class taught by $p$, then $p$ TEACHES $s$.

Userview — an alternative view on the application world.

A userview is a means of alternative comprehension of a part or all of the application world’s information. A userview consists of an alternative schema and inference rules by which every
An instantaneous database characterized by the original schema implies the (logical) instantaneous database characterized by the alternative schema (Fig. 1-19).

**Example 1-66.**
Figure 1-20 shows the alternative schema of the userview that covers the names of the courses and the seasons in which the courses are offered. It is more convenient to the user than the subschema of Figure ___.

![Diagram](image_url)

Figure 1-19. A userview.

COURSE SEASON

- name: String
- season: String m:m

One userview can be used by a subpopulation of the application world’s users. Such a userview would conceal from those users all the information which is irrelevant for them. The remaining information is presented to these users in a form which is most convenient to these particular users.

**Example 1-67.**
The computer science faculty secretary might use a userview containing only the addresses of the faculty. The userview has the alternative schema shown in Figure 1-21.
The inference rule for the category `COMPUTER-SCIENCE-INSTRUCTOR` is:

A computer science instructor is an `INSTRUCTOR` who `WORKS-IN` the `DEPARTMENT` whose `NAME` is "Computer Science."

```plaintext
PERSON
    last-name: String
    first-name: String
    birth-year: 1870..1990
    address: String

STUDENT

INSTRUCTOR

taught (m:m)
```

Figure 1-22. A userview: the inferred relation `taught`.

Some userviews do not omit any information and can be used by all the users. In this case, a userview presents the same information as the schema does, but in a form most suitable for some particular purposes.

All the information which can be deduced from an instantaneous database `idb` by a userview `u` is called the **instantaneous database under userview** `u` corresponding to `idb`.

Rishe-DDS p. 1.32
Unlike the schema, the alternative schema of a userview may contain redundant information if it adds to the convenience of the users. This redundancy cannot cause inconsistency, as it would in the case of the schema redundancy, since the updates are translated into the terms of the schema updates, before the updates are actually performed.

The usage of userviews also greatly enhances the flexibility of the database. Suppose we have a program that uses a userview, and the concepts of the application world change.

- If the change does not affect the logical decisions of this particular program, then we would like to avoid the need to modify the program.
- So, we will define a new userview to be used by the program.
- The alternative schema of the userview would be identical to the alternative schema of the old userview, so the program would not notice the difference.
- The inference rules would be, of course, different.
Figure 1-24. A change in the schema does not have to affect some users and programs.

Example 1-69.
Suppose we have some programs that use the information of the relation

\[ \text{works-in} \rightarrow \text{relation from INSTRUCTOR to DEPARTMENT (m:m)} \]

Now, the university has become more sophisticated, and for certain future programs it will be important to know the percentage of time that the instructors work for the departments. This new information is irrelevant to most old programs. The new schema will contain the fragment shown in Figure 1-25.

The alternative schema for the unaffected old programs will remain unchanged and will still contain the old relation WORKS-IN, but the inference rule of their new userview will be:

If there is a WORK whose department is \( d \) and whose instructor is \( i \), then \( i \) WORKS-IN \( d \).
**Note:** A subschema is a userview whose alternative schema is a part or all of the original schema, and the inference rules are trivial: they copy the information of the alternative schema’s concepts.

### 1.6. Services of DBMS

#### 1.6.1. Database languages: Query, DML, DDL, and others

This section introduces the major types of languages supported by database management systems.

**Retrieval query** — a specification of information which a user wants to extract or to deduce from the database without knowing the full extension of the instantaneous database.

**Example 1-70.**

The following is a retrieval query specified in English:

```
‘‘What students failed the Databases course last year?’’
```

Most DBMSs do not understand English. Thus, a specification in a more formal language is needed. Such a formal language need not be a programming language. Most DBMSs support some user-oriented formal language, in which the users can specify queries without writing programs.

**Query language** — a nonprogramming language in which a user can formulate *retrieval queries* and possibly also update the database. *Nonprogramming* means that the user does not have to specify an algorithm for the problem but only has to define the problem in a formal way.
Some query languages are simple enough to be used by the end user — a database user who has no computer knowledge or experience.

Other, more complex, languages are used by computer professionals. Yet, the professional user can save a significant amount of time by using such a nonprogramming language rather than writing a program.

**Data manipulation language, DML** — a programming language that has a powerful capability of computations, flow of control, input-output, and also has syntactic constructs for database access (the update, retrieval, and dynamic exchange of information between the program and the database). The DML is used by the application programmer.

A data manipulation language may be:

1. **A stand-alone DML** — a special-purpose language. In this case, the DBMS provides a compiler or an interpreter for the DML. The disadvantage of the stand-alone language is that it cannot be used for complex programs which perform some database access but also simultaneously perform other tasks, for example, numeric calculation or industrial assembly line monitoring.

2. **A system call interface.** In this case, the user writes a program in a regular programming language. The user performs database access by subroutine calls to the DBMS. In a call, the user provides the system with a description of the user’s request, parameters of the request, and output variables in which the system will produce the result of the database access.

```
Example 1-71.
The following is a fragment of a Pascal program with system calls.
...
    last-name := 'Jefferson'
    dbmscall (  
        'Dear DBMS: Please find the instructor whose last name is given in the second argument of this call. Place a reference to that instructor into the third argument of this call. If everything is OK, set the fourth argument of this call to 0. If there are several instructors by the given name, set the fourth argument to 1. If there are no such instructors, set it to 2. If another problem occurs, set the fourth argument to a number greater than 2.' (* In a real program, of course, a code would be given instead of this "short story." *),
        last-name, instructor-reference, return-code);
    if (return-code > 0)
        then write ('A DBMS error.')
    else begin
```

Rishe-DDS p. 1.36
The system-call interfaces are usually very unfriendly and hard to program in. The system-calls are interpreted at run-time of the program. One ramification of this is that if a system call contains a syntactically incorrect request, or a request inconsistent with the schema, the user cannot be notified at compile-time but has to wait until the program aborts. Then, the partial effects of the aborted program would have to be undone.

Another ramification is the poor efficiency of the run-time interpretation versus compilation.

3. **A DML embedded in a host programming language.** This is a database access extension of a general-purpose programming language. In a program, the host language statements are interleaved with DML statements.

**Example 1-72.**
The following is a two-statement fragment of an application program written in an embedded DML language whose host language is Pascal.

```pascal
write (`This is a regular Pascal statement which prints this sentence.');
relate: i WORKS-IN d (* This is a DML statement *)
```

The DBMS precompiles the program into a program in the host language without the DML statements. During the precompilation, the DBMS validates the syntax and the compatibility with the schema of the database. The DBMS may also perform optimization of the user’s algorithm.

The resulting program is compiled by the host language compiler. When the program runs, it may communicate to the DBMS, but the system calls of this communication are
transparent to the user.

**Report generator** — a language in which the user can specify a query together with requirements on the visual form of the output, such as nicely printed tables with titles, or bottom-of-page summaries.

**Data entry language** — a language in which the end users can specify database updates online.

**Data definition language, DDL** — a language in which the logical structure of information in the application’s world can be defined, together with its pragmatic interpretation for the management of a database, including the schema, integrity constraints, and userviews.

Many DDLs also allow for modification or redefinition of the database structure. This is needed when the concepts of the application world change and when the database designer finds a better description of the existing real-world concepts (particularly during the initial database design process, which is often trial and error).

In many DBMSs, the integrity constraints are specified in languages other than the DDL. In some DBMSs the constraint are incorporated in DML programs.

Many DBMSs do not provide any language at all to express the integrity constraints, other than those expressed in the schema. The other integrity constraints are left as the application programmer’s responsibility. In this case, an application programmer should write a DML program which will:

a. Collect the requests for updates from the users
b. Check whether these requests would violate the integrity constraints
c. Reject the requests that would violate the integrity
d. Submit to the DBMS the remaining good requests

### 1.6.2. Other services and utilities of DBMS

Most database management systems provide some or all of the following services and utilities.

1. **Integrity:**
   a. **Physical integrity** — prevention of a physical corruption of the database, such as placement of incorrect pointers or loss of an index when the power is shut off or the operating system fails.
   b. **Logical integrity** — enforcement of the integrity constraints.

2. **Backup and recovery:**
   a. The DBMS keeps backup copies of the database so that when the database is lost, or logically or physically corrupted, it can be restored to a previous state.
   b. The DBMS keeps a log of the updates it performs in the database so that when the database is restored to an old state, the correct updates kept in the log can be redone to make the new database up to date.

Rishe-DDS p. 1.38
c. When an application program is performing a complex update of the database, and the program gets aborted due to a run-time error or violates the integrity of the database or decides that it did not want that update after all, the DBMS can undo the update, that is, remove the partial effects of the update.

3. **Subschemas, userviews, inference rules** — see Section ___

4. **Security**
   
a. Prevention of persons who are not authorized database users from accessing the database. (This often involves verification of passwords or a similar procedure.)

b. Prevention of persons who are authorized users of a part of the database information from accessing other parts. This control is normally done through subschemas and userviews. A user may be allowed to read and update some information and to only read some other information. More complicated services include the distribution and revocation of information access privileges.

c. Encryption of the particularly sensitive physical files so that they can not be accessed by bypassing the DBMS.

5. **Concurrency control**:

   Several users and/or programs may access one database simultaneously. Some of the major problems addressed by the concurrency control are:

   a. An incorrect view of the database by one user during a database update being performed by another user.

   b. Inconsistent simultaneous updates being performed by different users.

   The logical side (the user’s perspective) of these problems is addressed in Section ___

6. **Data dictionary**:

   The information in the schema can be regarded as a small additional database. This database can be queried by the user.

   **Example 1-73.**
   
   A schema query:
   
   What relations have the category *STUDENT* as their domain?

   This additional database can also accumulate useful information which is not a proper part of the schema. This may include text explaining the informal meaning of every schema concept.

   **Example 1-74.**
   
   A data dictionary explanation entry for the concept ‘STUDENT’:

   *STUDENT* — "any person who is or was a registered student of the University at any time during the past 15 years, excluding persons who only attended short Extension courses of less than 5 days’ duration. ‘Registered’ means

Rishe-DDS p. 1.39
'paid the minimal registration fee at least once.' Insertion of a new student is initiated by the Office of the Registrar. Removal of a student from the database after the expiration of the 15-year period is performed automatically by the data manipulation program ARCHIVE, which is run every Sunday night. The removal of a student can also be explicitly performed by the Office of the Registrar as a correction of an error in inserting that nonstudent."

Some database management systems provide a "data dictionary" which can only assist the DML compiler in validating the program syntax. This is not a proper usage of the term because such a "dictionary" facility is no more than a minimal support of the schema, which is essential in any reasonable database management system.

The update of the data dictionary is normally a responsibility of the database administrator — a person who is the technical manager of a database.

7. Restructuring:
When the schema changes, the instantaneous database may become inconsistent with the schema. Restructuring is a utility to transfer the old instantaneous database into a new instantaneous database under a new schema.

8. Distributed database:
The database of a large enterprise may be physically partitioned into several subsets which are stored in different geographical sites, such as the branches of the enterprise. In order to avoid the costs and delays of telecommunication, each site physically contains the information which is most frequently used in that site.

Example 1-75.
Assume that our University has several campuses. In every campus we would physically store information most frequently used in that campus:

- a. The departments of that campus
- b. The instructors who work in those departments
- c. The students whose majors or minors are among those departments
- d. The offerings of courses by those instructors
- e. The enrollments in those offerings

Logically, the users at each site see the whole database (or the fractions thereof which they are authorized to see) regardless of the database’s partition. When a user’s query or update request necessitates the access to information which is beyond the user’s site, the system performs this access in a way transparent to the user.

Rishe-DDS p. 1.40
Example 1-76.
Assume that there is a student $a$ at campus $c_1$ who took some courses offered at another campus, $c_2$, by instructors of that campus $c_2$.

Now, an administrator of Campus $c_1$ submits a query:

‘Calculate the grade-point-average of Student $s$.’

The administrator need not be aware of the physical distribution of the information. During the execution of the query, the DBMS will decide that it needs some information from Campus $c_2$, contact that campus’s computer, and give the user the correct result as if all the data were available locally.

The physical subsets comprising the distributed database do not have to be disjoint.

Example 1-77.

a. The personal information about a student whose major is at one campus and whose minor is at another campus will be stored in both campuses. The enrollments of that student do not have to be duplicated, since enrollments are stored according to the offerings, which, in turn, are stored according to instructors, which are stored by departments.

b. The information about an instructor who works in Department $d_1$ of Campus $c_1$ and also in Department $c_2$ of Campus $c_2$ will be stored at both campuses.

c. The catalog of the names of all the courses can be duplicated at each campus, since it is frequently accessed at each campus.

The possible physical duplication of information introduces one of the major problems of the distributed database management: how to maintain consistency between the copies of information.

One of the other problems is the optimization of the routing of information between the sites during the execution of queries and update requests. Another problem is tolerance for failures: when one site or one communication line goes down, we do not want to shut down the whole system.

1.7. Multimedia Databases

If the DBMS has no implementational restrictions as to the sizes of strings, then the database can store multimedia data as subcategories of the concrete category String. The following are some of such subcategories:
Text — arbitrarily long texts (e.g. the entire text of a book).

Image — digitized color photograph.

Audio — digitized speech or music.

**Line-drawing** — representation of diagrams, signatures, etc.

<table>
<thead>
<tr>
<th>Example 1-78.</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐ syllabus — attribute of COURSE, range: Text (m:1)</td>
</tr>
<tr>
<td>☐ photo — attribute of PERSON, range: Image (m:1)</td>
</tr>
<tr>
<td>☐ signature — attribute of INSTRUCTOR, range: Line-drawing (m:1)</td>
</tr>
<tr>
<td>☐ outgoing-message — attribute of INSTRUCTOR, range: Audio (m:1) (The message to be played by the telephone system when the instructor being called is not available.)</td>
</tr>
</tbody>
</table>

**Reference Schemas**

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.
Figure Ref-1. A semantic schema for a university application.
Figure Ref-2. A relational schema for the university application.
2. SEMANTIC LANGUAGES

Two abstracted languages, or rather language models for comprehending specific languages, are presented here:

- A fourth-generation structured extension of a structured third-generation programming language (Pascal taken as an example)
- A nonprocedural predicate-calculus language

The languages are defined here for the Semantic Binary Model. Since all the other major database models will be defined as subsets of the Semantic Binary Model, these languages will be used for all the models.

Many languages, similar to one of the two languages studied here, with some syntactic variations, are in use in different database models and different database management systems. The purpose of the presentation here is to delineate the common denominators of the classes of languages, while avoiding the technical details specific to particular systems.

2.1. Notation

The following syntactic notation is used in the program fragments and in the definitions of syntactic constructs:

- Language keywords are set in **boldface**.
- The names of the relations and categories from the database are set in **UPPERCASE ITALICS**.
- In syntax description templates, items to be substituted are set in *lowercase italics*.

Example 2-1.

- **procedure** — a language keyword
- **LAST-NAME** — a database relation
- **expression** — in a syntax template, substitute for an actual expression, for example, 
  
  "(7+8*x)"

Gender: the pronoun *he* includes *she* and the pronoun *she* includes *he*.  

Rishe-DDS p. 2.1
2.2. Fourth-Generation Programming

2.2.1. Principles

Programming languages which exhibit well-structured flow of control, elaborate typing, and a high degree of machine independence are called third-generation programming languages. An example of such a language is Pascal. In this section, we shall use the example of Pascal to introduce the principles of fourth-generation database manipulation extensions of third-generation programming languages.

The essence of the fourth-generation data manipulation languages is the **structured access to the database**. This is contrasted with earlier data manipulation languages, which provided no automatic loops to process bulks of information in the database but only single commands to access one item at a time. As a result, the programmer was left with the responsibility of "navigating" between different data items in the database.

The principal instruction of the language extension to be introduced is the **for** loop, whose body is executed for every object which is present in the instantaneous database (when the program is run) and satisfies conditions given in the **for** statement.

**Example 2-2.**

(* Print the name of every instructor, that is, of every object in the category INSTRUCTOR. *)

```pascal
for instructor in INSTRUCTOR do
  (* Print the name of the current instructor, that is, of the object referred to by the variable instructor. Separate the current name by a blank ' ' from the name printed in the previous iteration of this loop. *)
  write (' ', instructor.LAST-NAME)
```

**Example 2-3.**

(* Print the name of every Computer Science instructor, that is, of every object in the category INSTRUCTOR who WORKS-IN the department whose name is ‘Computer Science’. *)

```pascal
for department in DEPARTMENT
  where (department NAME ‘Computer Science’)
  do
    for instructor in INSTRUCTOR
      where (instructor WORKS-IN department)
      do
        (* Print the name of the current instructor, that is, of the
```

Rishe-DDS p. 2.2
2.2.2. Specification of Extended Pascal

The following is a fourth-generation extension of Pascal for structured access to databases.

2.2.2.1. Data types and parameters

1. **Global parameters** — among the global parameters of a program such as INPUT and OUTPUT, there are the names of the database and of the userview. The database will be accessed through the userview during the execution of the program. The userview will also be accessed during the compilation of the program in order to check for the correct usage of the names of the categories and relations and to correctly interpret the program’s commands.

   Example 2-4.

   Let UNIVERSITY-MASTER-VIEW be the userview identical to the whole schema. The following may be an Extended Pascal heading for a program using the whole schema of the University database. We assume that the name of the database is UNIVERSITY-DB. This name will be used by the DBMS to locate all the files comprising the database.

   ```pascal
   program My-program (Input, Output, UNIVERSITY-DB, UNIVERSITY-MASTER-VIEW);
   ```

2. **Data type** ABSTRACT — a new basic data type, in addition to INTEGER, BOOL, REAL, CHAR, enumerated types, and STRING. (The type STRING is not defined in the standard Pascal but is used, sometimes with a different name, in most practical versions of Pascal.)

The variables of type ABSTRACT will contain abstract objects. (Practically, these variables will contain logical references to abstract objects. The referencing, however, is transparent to the user.) The variables of this type are called abstract variables.

The abstract variables cannot be printed. They cannot receive a value through a read instruction. There are no constants of type ABSTRACT.

Assignment to the abstract variables can be done from other abstract variables or from the database or by the instruction create as discussed later.

In expressions, the only meaningful operation on arguments of type ABSTRACT is the test for their equality. The equality test, ";=", produces TRUE if the two arguments are one and the same object in the database.
Example 2-5.

```pascal
var jackson: ABSTRACT (* The abstract variable jackson may be used to retrieve from the database a reference to Professor Jackson, that is, to the abstract object of the category INSTRUCTOR related by the relation LAST-NAME to the string ‘Jackson’. *)
```

2.2.2.2. Extended expressions

There are new operators which can be used in Pascal expressions:

1. \((expression-of-type-ABSTRACT \text{ is a } \text{category-from-the-userview})\)

   This Boolean expression gives TRUE when the left-side subexpression is evaluated into an object which is a member of the category on the right side.

   The membership test is done according to the information in the instantaneous database at the run time of the program.

   **Example 2-6.**
   
   ```pascal
   · (jackson \text{is a } STUDENT)
   ```

   If the variable jackson is uninitialized, then a run-time error results. If this variable contains an abstract object, then the result of the expression is TRUE if that object is a student. If this object is simultaneously a student and an instructor, the result is still TRUE.

2. \((expression \text{ relation-from-the-userview expression})\)

   This Boolean expression gives TRUE when the two subexpressions yield objects participating in the relation in the instantaneous database. The types of the subexpressions must be consistent with the relation. For example, if the relation is between abstract objects and real numbers, then the type of the left subexpression must be ABSTRACT and the type of the right subexpression must be REAL.

   **Example 2-7.**
   
   ```pascal
   · (jackson FIRST-NAME ‘Roberta’)
   · (jackson BIRTH-YEAR 1960)
   ```

   Instead of one of the subexpressions, the keyword null may appear. Then the Boolean expression would give TRUE if the object yielded by the remaining subexpression is related by the relation to no object in the instantaneous database.

Rishe-DDS p. 2.4
Example 2-8.

- (jackson WORKS-IN null)

This expression yields TRUE when the person referred to by the variable jackson does not work in any department.

Example 2-9.

- (jackson BIRTH-YEAR null)

This expression yields TRUE if, for the person referred to by the variable jackson, no birth-year was recorded in the database.

3. (expression, functional-relation-from-the-userview)

Reminder: a functional relation is an m:1 relation. It relates every object of its domain to at most one object of its range.

The expression $x.R$ produces the object related by the relation $R$ to $x$; that is, the result is the object $y$ from the instantaneous database such that $(x \ R \ y)$ is TRUE.

If no such object $y$ exists, then a null object results, which can cause a subsequent execution-time error.

Example 2-10.

- (jackson. BIRTH-YEAR)

Example 2-11.

Here is a program fragment to print the age of the person referred to by the variable jackson. We assume that the current year is available in the variable current-year.

```plaintext
write ('Professor ', jackson.FIRST-NAME, ',', jackson.LAST-NAME, ' is approximately ', (current-year - jackson.BIRTH-YEAR), ' years old.')
```

2.2.2.3. Atomic database manipulation statements

1. create new abstract-variable in abstract-category-from-the-userview

- A new abstract object is created in the database
- This object is placed into the specified category (the database is updated to reflect this fact)
• [A reference to] this object is assigned to the specified variable

**Example 2-12.**

```java
var department: ABSTRACT; ...
create new department in DEPARTMENT
```

This instruction has two effects:

• A new abstract object is created in the category *DEPARTMENT* in the database.

• A reference to this object is assigned to the variable *department* in the program’s memory.

2. **categorize:** *expression-of-type-ABSTRACT is a category*

The expression is evaluated to produce an *existing* instantaneous database object, and this object is inserted into the specified category (in addition to other categories the object may be a member of).

**Example 2-13.**

Let the variable *jackson* refer to an existing instructor Professor Jackson. The following instruction will place Professor Jackson also into the category *STUDENT* (in addition to the category *INSTRUCTOR*).

```java
categorize: jackson is a STUDENT
```

This instruction has only one effect: a change in the instantaneous database. It produces no change in the program’s working space; that is, it does not change the contents of any variable.

3. **decategorize:** *expression-of-type-ABSTRACT is no longer a category*

The object is removed from the category.

The object is also automatically removed from the subcategories of the category. (Otherwise the database would become inconsistent.)

The object is also automatically removed from the relations whose domains or ranges are categories of which the object is no longer a member. (This automatic removal saves programming effort. This removal is also necessary to maintain the consistency of the database.)

If after the decategorization the object would not belong to any category in the database, then the object is removed from the database.
Example 2-14.

decategorize: jackson is no longer a STUDENT

The person referred to by the variable jackson will no longer be a student. She
will no longer participate in any relation whose domain or range is the category
STUDENT. For example, she will be disconnected from her major and minor
departments.

Example 2-15.

The following instruction removes the object referred to by the variable jackson
from the database:

decategorize: jackson is no longer a PERSON

4. relate: expression relation expression

A new fact is added to the database: a relationship between the objects yielded by the
two expressions.

Example 2-16.

Assuming that the variable jackson refers to an instructor whose birth-year was
not known until now, the following instruction will set the birth-year:

relate: jackson BIRTH-YEAR 1961

Example 2-17.

Assuming that the variable jackson refers to an instructor who is also a student
having a major department, the following instruction will make Jackson work in
her major department. If she was also working in some other department, she
will continue working there too.

relate: jackson WORKS-IN jackson.MAJOR

5. unrelate: expression relation expression

This has the reverse effect of the instruction relate.

Example 2-18.

Assuming that the variable jackson refers to an instructor whose birth-year has
been incorrectly recorded, the following instructions will change the birth-year to
1961:
unrelate: jackson \textit{BIRTH-YEAR} jackson.BIRTH-YEAR

(* The expression "jackson.BIRTH-YEAR" gives the previously recorded birth-year.*);

relate: jackson \textit{BIRTH-YEAR} 1961

6. \textit{expression}.\textit{relation} := \textit{expression}

The assignment statement

\[ x.R := y \]

means:

- For every \( z \), unrelate \( x R z \)
- Then relate \( x R y \).

\textbf{Example 2-19.}

Assuming that the variable \textit{jackson} refers to an instructor whose birth-year has not been recorded yet, or has been incorrectly recorded, the following instruction will make the birth-year 1961:

\[ \text{jackson.BIRTH-YEAR} := 1961 \]

\textbf{Example 2-20.}

Assume that the variable \textit{math} refers to the Mathematics department, and the variable \textit{miller} refers to an instructor. What is the effect of the following instruction?

\[ \text{miller.WORKS-IN} := \text{math} \]

Miller will \textit{only} be working in the Mathematics department.

- If Miller was not working in any department, he will be working in Mathematics.
- If Miller was working in the Management Science and Physics departments, he is hereby fired from Management Science and Physics, and hired in Mathematics.
2.2.2.4. The for statement

The for statement is the core of fourth-generation programming. This statement creates a structured loop. Syntax:

```
for variable in category
   where boolean-expression
   do statement
```

**Interpretation:**

The `statement` after `do`, which may be a compound statement, will be performed once for every object which belongs to the `category` and satisfies the `boolean-expression`.

**Example 2-21.**

Print the last names of all the students born in 1964.

```
for s in STUDENT
   where s BIRTH-YEAR 1964
   do writeln(s.LAST-NAME)
```

**Example 2-22.**

Print the last names of all the students.

```
for s in STUDENT
   where true
   do writeln(s.LAST-NAME)
```

The for statement is functionally equivalent to the following algorithm.

Let `VEC` of length `L` be the vector of all the `category`’s objects in the instantaneous database. The vector is arranged in an arbitrary order, transparent to the user. Then the equivalent algorithm for the for statement is:

```
for i := 1 to L do
   begin
      variable := VEC [i];
      if boolean-expression
      then statement
   end
```

**Abbreviation:**

In the for statement, the “in category” part may be omitted. In this case, by default the category is assumed to be a special category `OBJECT`, which is regarded as the union of all the abstract categories in the database. Thus, the body of the loop will be executed for every abstract object (in the instantaneous database) satisfying the condition of the loop. Practically, the condition may explicitly or implicitly restrict the loop to one
category.

**Example 2-23.**

Print the last names of all the persons born in 1964.

```pascal
for s where s.BIRTH-YEAR 1964 do
  writeln (s.LAST-NAME)
```

**Example 2-24.**

A larger program fragment:

Who are the persons that taught persons that taught persons that taught persons that taught Mary?

```pascal
for mary in STUDENT where (mary.FIRST-NAME 'Mary') do
  for enrl1 in COURSE-ENROLLMENT where (enrl1.THE-STUDENT mary) do
    for enrl2 in COURSE-ENROLLMENT where (enrl2.THE-STUDENT
      enrl1.THE-OFFERING.THE-INSTRUCTOR) do
      for enrl3 in COURSE-ENROLLMENT where (enrl3.THE-STUDENT
        enrl2.THE-OFFERING.THE-INSTRUCTOR) do
        for enrl4 in COURSE-ENROLLMENT where (enrl4.THE-STUDENT
          enrl3.THE-OFFERING.THE-INSTRUCTOR) do
          writeln (enrl4.THE-OFFERING.THE-INSTRUCTOR.LAST-NAME)
```

2.2.2.5. Transactions

1. The transaction statement

```pascal
transaction compound-statement
```

The effects of "transaction S" are:

a. While S is being executed, the program containing the transaction statement and all
   the other concurrent programs see the database in its instantaneous state just before
   S.

b. All the updates are logically performed instantly when S is completed, provided the
   new instantaneous database would not violate the integrity constraints and no
   error-condition is raised.

Rishe-DDS p. 2.10
Note:
Among the advantages of this statement is the following:

At an intermediate state, the instantaneous information could be incomplete, which
could bring failure of an integrity constraint and incorrect comprehension of the
database by concurrent programs.

Example 2-25.
Assume that the relation LAST-NAME is total. We wish to create a new person
and give her the last name ‘Chen’.

The following is a wrong program fragment to perform the task:

```plaintext
create new chen in PERSON;
chen.LAST-NAME := ‘Chen’
```

The above program would violate the integrity of the database when the
instruction create was performed: there would be a person with no last name,
contrary to the totality of the relation LAST-NAME. The program would probably
be aborted before it reached the assignment statement. To prevent the integrity
validation between the two statements, we enclose them in a transaction
statement:

```plaintext
transaction begin
    create new chen in PERSON;
    chen.LAST-NAME := ‘Chen’
end
```

Also, we would not have to worry that concurrent queries see the database half-
updated: a person without a last name.

Example 2-26.
The database needs to be changed to pretend that the person referred to by the
variable jackson is not, nor has ever been, a student. Thus, the object has to be
decategorized from STUDENT, and all the relevant enrollments have to be erased.

```plaintext
transaction
    begin
        for e in COURSE-ENROLLMENT where (e THE-STUDENT jackson) do
            decategorize: e is no longer a COURSE-ENROLLMENT;
            decategorize: jackson is no longer a STUDENT
    end
```

Rishe-DDS p. 2.11
A database update statement which is not embedded in a transaction statement is regarded as one transaction.

2. **Error exit**

When the system fails to perform a transaction because of an error, such as a violation of an integrity constraint, it notifies the program by invoking

```
procedure Transaction-error-handler (error-description: String)
```

The body of this procedure can be specified in the program by the user. This allows the programmer to decide what to do in case of error. If the procedure is not defined by the user in the program, then, by default, the system will insert the following specification of the body of this procedure:

```
procedure Transaction-error-handler (error-description: String);

begin
  writeln ('The program was terminated by the default transaction error handler when a transaction failed with the following error condition: ', error-description);

  stop
end
```

*Example 2-27.*

The user can specify the following handler, which prints a message and then allows the program to continue.

```
procedure Transaction-error-handler (error-description: String);

begin
  writeln ('A transaction failed with the following error condition: ', error-description);

end
```

### 2.2.3. A programming example

*Example 2-28.*

The university has decided to expel all the students whose average grade is below 60 (out of 100). To prevent this wrongdoing to computer science students, the department offered a fictitious course, Computer-Pass, by Prof. Good, in which all computer science students are to receive a sufficient grade so as to not to be expelled, if possible.

The following program fabricates Prof. Good and the Computer-Pass
course, enrolls students in this course, grades them accordingly, and prints the
names of those computer science students whom this measure cannot help.

program Pass (Input, Output, UNIVERSITY-DB, UNIVERSITY-MASTER-VIEW);
var Computer-Pass-Course, Prof-Good, Good-Offer, comp-science, this-quarter, cs-student,
er-enrollment, fictitious-enrollment: ABSTRACT;
desired-grade, number-of-grades, total-of-grades, current-year: INTEGER;

begin
  (* Get the current year from the standard input file. *)
  read (current-year);
  (* Fabricate the course. *)
  create new Computer-Pass-Course in COURSE;
  Computer-Pass-Course.NAME := ‘Computer Pass’;
  (* Fabricate the instructor. *)
  create new Prof-Good in INSTRUCTOR;
  Prof-Good.LAST-NAME := ‘Good’;
  (* Fabricate the offering. *)
  create new Good-Offer in COURSE-OFFERING;
  Good-Offer.THE-COURSE := Computer-Pass-Course;
  Good-Offer.THE-INSTRUCTOR := Prof-Good;
  (* Find the relevant quarter and connect it to the offering Good-Offer. *)
  for this-quarter in QUARTER
    where (this-quarter.YEAR = current-year and this-quarter.SEASON = ‘Winter’)
    do Good-Offer.THE-QUARTER := this-quarter;

  (* The following loop will be performed only once. Inside the body of the loop, the variable
  comp-science will refer to the Computer Science Department. *)
  for comp-science in DEPARTMENT
    where (comp-science NAME ‘COMPUTER SCIENCE’) do
      begin
        (* Make believe that Prof. Good works in Computer Science. *)
        relate: Prof-Good WORKS-IN comp-science;

Rishe-DDS p. 2.13
for cs-student in STUDENT
    where (cs-student MAJOR comp-science) do
    begin (* the current computer science student *)
        (* Calculate this student’s current statistics: number-of-grades and total-of-grades *)
        number-of-grades := 0;
        total-of-grades := 0;
    for her-enrollment in COURSE-ENROLLMENT
        where (her-enrollment THE-STUDENT cs-student) do
            if not (her-enrollment FINAL-GRADE null) then
                begin
                    number-of-grades := number-of-grades + 1;
                    total-of-grades := total-of-grades + her-enrollment.FINAL-GRADE;
                end;
        (* calculate the minimal desired grade in computer-pass course, by solving the equation (total+x)/(number+1)=60 *)
        desired-grade := 60 * (number-of-grades + 1) — total-of-grades;
    if desired-grade > 100 then
        (* the student cannot be helped. Print a message *)
        writeln (‘ The student ’, cs-student.LAST-NAME, ‘ cannot be helped. Sorry!’)
    else if desired-grade ≤ 60 then
        (* No need to help. *)
    else (* 100 ≥ desired-grade > 60 *)
        transaction begin
            create new fictitious-enrollment in COURSE-ENROLLMENT;
            fictitious-enrollment.THE-OFFERING := Good-Offer;
            fictitious-enrollment.THE-STUDENT := cs-student;
            fictitious-enrollment.FINAL-GRADE := desired-grade
        end (* transaction *)
    end (* current student *)
end (* Computer Science Department *)
end.
2.3. Logic as a Nonprocedural Language

2.3.1. Principles

Nonprocedural language — a language in which the user specifies what is to be done without specifying how it is to be done.

Example 2-29.

In a nonprocedural language, the user might say:

```
‗‗Let no student be enrolled twice in the same offering of a course‘‘
```

The user would probably use a more precise and formal statement, which still would be nonprocedural:

```
‗‗If enrl1 is an enrollment, and enrl2 is an enrollment, and
enrl1.THE-STUDENT = enrl2.THE-STUDENT, and
enrl1.THE-OFFERING = enrl2.THE-OFFERING,
then enrl1 = enrl2.‘‘
```

This statement contains no indication as to how the constraint is to be enforced: this is left to the system. The system might enforce the constraint as follows:

Whenever a transaction like

```c
transaction begin ... 
create new enrl in COURSE-ENROLLMENT;
enrl.THE-STUDENT := s;
enrl.THE-OFFERING := of;
...
end
```

is being completed, perform automatically the following:

`(* If there is another enrollment with the same student and offering, then give an error message and stop. *)`

```c
for other-enrl in COURSE-ENROLLMENT
where ( other-enrl.THE-STUDENT = s and other-enrl.THE-OFFERING = of)
do begin
writeln (‘ In violation of an integrity constraint, the program has attempted to generate a duplicate enrollment’); stop
end
```

Rishe-DDS p. 2.15
Example 2-30.

In a nonprocedural language, the user might request the following information:

```
“What instructors give the grade 100 (sometimes)?”
```

The user would probably use a more precise and formal statement, which still would be nonprocedural:

```
“Get instructor.LAST-NAME where exists enrollment such that enrollment.THE-OFFERING.THE-INSTRUCTOR = instructor and enrollment.FINAL GRADE = 100”
```

The statement would contain no indication as to how this query is to be performed: this is left to the system. The system might use the following algorithm:

```pascal
for instructor in INSTRUCTOR
do begin
  ok := false
  for enrl in COURSE-ENROLLMENT
    where (enrl.THE-OFFERING.THE-INSTRUCTOR = instructor and enrl.FINAL GRADE = 100)
    do ok := true
  if ok then write (‘ ‘, instructor.LAST-NAME)
end
```

Many nonprocedural database languages are based on Predicate Calculus, borrowed from Mathematical Logic. The language of Predicate Calculus will be defined in the following subsection.

*Uses*: nonprocedural specification of queries, integrity constraints, inference of userviews, and update transactions.

Predicate Calculus is based on Boolean expressions involving variables.

Example 2-31.

- instructor.LAST-NAME = ‘Einstein’
- d NAME  ‘Geology’
- student.BIRTH-YEAR > 1970 or student.BIRTH-YEAR = 1960
Such Boolean expressions can be used to specify queries similarly to the specification of sets in mathematics:

- A set of objects is found which make a Boolean expression to be true.
- These objects or their functions are displayed.

**Example 2-32.**
A query to find the names of the persons born in 1967:

```sql
get person.LAST-NAME
where (person.BIRTH-YEAR = 1967)
```

A query can display several columns of output, in a table form.

**Example 2-33.**
A query to find the first and last names of the persons born in 1967:

```sql
get person.FIRST-NAME, person.LAST-NAME
where (person.BIRTH-YEAR = 1967)
```

Several different objects, referred to by different variables, may be used in one row of the output tables.

**Example 2-34.**
For every student, list the instructors of the student’s major department.

```sql
get student.FIRST-NAME, student.LAST-NAME, instructor.FIRST-NAME, instructor.LAST-NAME
where (instructor WORKS-IN student.MAJOR)
```

Very powerful constructs of Predicate Calculus are the quantifiers for every and exists. They are used to form nontrivial Boolean expressions.

**Example 2-35.**
What instructors work in every department? (Each relevant instructor shares her time between all the departments.)

```sql
get instructor.LAST-NAME where
```

Rishe-DDS p. 2.17
(for every d in DEPARTMENT:
instructor WORKS-IN d)

Example 2-36.
What instructors taught every student?
get instructor.LAST-NAME where
(for every s in STUDENT:
exists enrl in COURSE-ENROLLMENT:
((enrl THE-STUDENT s) and
(enrl.THE-OFFERING. THE-INSTRUCTOR = instructor)))

2.3.2. First-order predicate calculus expressions

The First-order Predicate Calculus is well-known to those who have studied some Mathematical Logic. This calculus can be applied to databases, if we regard the instantaneous database as a finite structure with binary relations, unary relations (categories), and functions (functional relations). This text, however, does not require a prior knowledge of Predicate Calculus.

Expression — a combination of constants, variables, operators, and parentheses. The syntax and semantics are given below.

An expression may depend on some variables. When the variables are interpreted as some fixed objects, the expression can be evaluated with respect to a given instantaneous database and will yield an abstract or concrete object. The following are syntactic forms of expressions:

1. constant
   a. number
   b. character-string (in quotes)
   c. Boolean value (TRUE and FALSE)

Example 2-37.
7, 16.5, ‘Mary’, ‘87/05/31’, TRUE

2. variable
   A variable is a sequence of letters, digits, and hyphens. The first character must be a letter.

Rishe-DDS p. 2.18
Parentheses in expressions may be omitted when no ambiguity results.

The basic binary operators are: +, -, *, /, >, ≥, ≤, =, ≠, and, or.

Each operator may be used only when the expressions yield values of types appropriate for the operator. The only basic binary operators defined for abstract objects are = and ≠, which produce TRUE or FALSE as results.

Example 2-38.
- 5 + 6 * 7
- x ≠ y
- (`Abc’ > ‘Bcc’) or (1 + 2 > 2)

Both component expressions must yield Boolean values. When the right expression yields TRUE, the result is TRUE regardless of the left expression. When the left expression yields TRUE and the right expression yields FALSE, the result is FALSE. When the left expression yields FALSE, the result is TRUE regardless of the right expression.

Example 2-39.
The following expressions are TRUE:
- if 1=1 then 2=2
- if 1=3 then 2=2
- if 1=3 then 2=4

The following is FALSE:
- if 1=1 then 2=4

Note:
if \( e_1 \) then \( e_2 \)
is equivalent to
\((\text{not } e_1) \text{ or } e_2\)

The relation is a relation from the userview. The result is TRUE if the two objects are related by the relation in the instantaneous database.
Example 2-40.
\[(x \text{ BIRTH-YEAR } 1960)\]
The value of this Boolean expression depends on the variable \(x\).

7. \(\text{(basic-unary-operator expression)}\)
The basic unary operators are: \(-\), \(\text{not}\).

Example 2-41.
\[(\text{not} (1>1)) = \text{TRUE}\]

8. \(\text{(expression is a category)}\)
This Boolean expression yields \text{TRUE} when the object is in the category in the instantaneous database.

Example 2-42.
\(x\ \text{is a STUDENT}\)

9. \(\text{(expression . functional-relation)}\)
\(x.R\) is the object related by the relation \(R\) to \(x\); it is the object \(y\) from the instantaneous database such that \((x \ R \ y)\) gives \text{true}. Such an expression is called \text{dot-application}.

Example 2-43.
- \(x.BIRTH-YEAR\)
- \(e.THE-OFFERING.THE-INSTRUCTOR.LAST-NAME\)

The dot-application is well-defined only for total functional relations. The case of nontotal functional relations will be discussed later.

10. \(\text{(exists variable in category : expression)}\)
The ‘::’ may be pronounced ‘so that the following is true:’.
The contained expression must be Boolean.
The result is also Boolean.
It is \text{TRUE} when there exists at least one object in the category which satisfies the Boolean expression.
The expression usually depends on the variable but may also depend on additional variables. The resulting expression no longer depends on the variable.
Interpretation:

Let \( a_1, a_2, \ldots, a_n \) be all the objects in the category in the instantaneous database.

Let \( e_1, e_2, \ldots, e_n \) be obtained from the expression by substituting each of \( a_1, a_2, \ldots, a_n \) for all the occurrences of the variable in the expression.

Then \( \text{exists variable in category : expression} \)

is equivalent to \( e_1 \text{ or } e_2 \text{ or } \cdots \text{ or } e_n \)

Example 2-44.

(\text{exists } x \text{ in INSTRUCTOR} : \\
\quad \quad x.BIRTH-YEAR = 1960)

This expression is TRUE if there is at least one instructor who was born in 1960. The whole expression does not depend on the variable \( x \), although its subexpression "\( x.BIRTH-YEAR = 1960 \)" does depend on this variable.

Example 2-45.

(\text{exists } x \text{ in INSTRUCTOR} : \\
\quad \quad x.BIRTH-YEAR = y)

This is TRUE if there is at least one instructor who was born in the year \( y \). The whole expression depends only on the variable \( y \).

The keyword \text{exists} is often called the \textbf{existential quantifier}. It may be abbreviated by the symbol \( \exists \).

11. \textbf{(for every variable in category : expression)}

The ‘:’ is pronounced ‘the following is true:’.

The expression must be Boolean. The result is also Boolean. It is TRUE when all the objects of the category satisfy the Boolean expression. The expression usually depends on the variable and may also depend on additional variables. The resulting expression no longer depends on the variable.

Interpretation:

Let \( a_1, a_2, \ldots, a_n \) be all the objects in the category in the instantaneous database.

Let \( e_1, e_2, \ldots, e_n \) be obtained from the expression by substituting each of \( a_1, a_2, \ldots, a_n \) for all the occurrences of the variable in the expression.

Then
for every variable in category : expression

is equivalent to

\[ e_1 \text{ and } e_2 \text{ and } \cdots \text{ and } e_n \]

**Example 2-46.**

(for every x in INSTRUCTOR :  
\[ x.BIRTH-YEAR = 1960 \])

This expression is TRUE if all the instructors were born in 1960.

Notice that the result is TRUE even if there are no instructors at all in the instantaneous database. (Isn’t it true that every presently living dinosaur knows Predicate Calculus or that every American monarch is a republican?)

The whole expression does not depend on the variable \( x \), although its subexpression "\( x.BIRTH-YEAR = 1960 \)" does depend on this variable.

**Example 2-47.**

(for every x in INSTRUCTOR :  
\[ x.BIRTH-YEAR = y \])

This is TRUE if all the instructors were born in the year \( y \). The whole expression depends only on the variable \( y \).

The keyword for every is often called the universal quantifier. It may be abbreviated by the symbol \( \forall \).

Note:

for every variable in category : expression

is equivalent to

not (exists variable in category : not expression)

**Usage of variables:**

The variable after a quantifier in a subexpression should not be used outside that subexpression. Although many versions of Predicate Calculus do not have this requirement, it does not decrease the power of the calculus but improves readability, prevents some typical errors in query specification, and simplifies the semantics.

**Example 2-48.**

WRONG:  
(exists x in PERSON: x is a STUDENT) and (x BIRTH-YEAR 1970)

Here, \( x \) appears in the quantifier of the left subexpression but also appears in the
right subexpression. Logically, these are two distinct variables, and they should not be called by the same name $x$.

To use the expressions correctly, we shall need to know what variables are quantified in an expression and on what variables an expression depends.

**Quantified variable** — variable $v$ is quantified in expression $e$ if $v$ has an appearance in $e$ immediately after a quantifier.

*Example 2-49.*
The variable $v$ is quantified in:

$$((z > 0) \text{ or } \exists v \text{ in STUDENT: } v \text{ is an INSTRUCTOR})$$

Expression $e$ depends on variable $v$ if $v$ appears in $e$ and is not quantified.

*Example 2-50.*
The following expression depends on $z$ and $x$ but not on $y$.

$$((z > 0) \text{ or } \exists y \text{ in STUDENT: } x = y.BIRTH-YEAR)$$

**Notation:** When an expression $e$ depending on variables $x_1, x_2, \ldots, x_k$ is referred to (not in the actual syntax of the language), it may be denoted as $e(x_1, x_2, \ldots, x_k)$

*Example 2-51.*
The expression of the previous example may be referred to as $e(z, x)$.

(In many texts, a variable on which an expression depends is called a **free variable** in that expression. An expression which depends on no variables is called a **closed expression**.)

**Condition** on variables $x_1, x_2, \ldots, x_k$ — a Boolean expression which depends on $x_1, x_2, \ldots, x_k$.

*Example 2-52.*

$(x+y > 3)$ is a condition on $x$ and $y$.

**Assertion** — a Boolean expression which does not depend on any variable; that is, every variable is restricted by a quantifier.
Interpretation: For a given instantaneous database, the assertion produces TRUE or FALSE.

Example 2-53.
Assertion that every student took at least one course in 1990:

for every st in STUDENT:
    exists enrl in COURSE-ENROLLMENT:
        ((enrl.THE-STUDENT st ) and

Dot-application of nontotal functional relations
If \( f \) is not total, then \( e.f \) may be ambiguous. The concerned user might wish to avoid such expressions. However, a smart DBMS may be able to follow the user’s intuition in using such expressions. This is discussed further in Section ___

2.3.3. Queries

Specification of a query to retrieve a table, that is, a set of rows of values:

get expression, . . . , expression
where
(condition-on-the-variables-on-which-the-expressions-depend)

Interpretation of the query get \( e_1, \ldots, e_n \) where \( \phi(x_1, \ldots, x_k) \)

- The variables \( x_1, \ldots, x_k \) are assigned all the possible tuples of objects which objects are in the current instantaneous database and which make the assertion \( \phi(x_1, \ldots, x_n) \) evaluate to TRUE.
- The expressions \( e_1, \ldots, e_n \) are evaluated for the above selected tuples and the corresponding results are output. (The output is not printable if any of the expressions produces an abstract object.)

Example 2-54.
Who took Prof. Smith’s courses?

get student.LAST-NAME where
exists enrl in COURSE-ENROLLMENT:
    (enrl.THE-STUDENT=student and
    enrl.THE-OFFERING. THE-INSTRUCTOR. LAST-NAME=‘Smith’)
**Abbreviation:**

The following abbreviation is used in some literature for the query syntax. It is akin to the mathematical notation for sets.

\[
\{e_1, \ldots, e_n \mid \phi(x_1, \ldots, x_k)\}
\]

**Abbreviation:**

*Queries which output only one value* may be specified without the ‘‘where condition’’ part, as:

```
get expression
```

(provided the expression depends on no variables).

---

**Example 2-55.**

The following is a yes-or-no query which displays TRUE if every student took at least one course.

```
get

(for every s in STUDENT:
    exists enrl in COURSE-ENROLLMENT:
        s=enrl.THE-STUDENT)
```

---

**Headings of output columns:**

The columns in a table which is an output of *get* can be labeled:

```
get heading_1: e_1, \ldots, heading_n: e_n where condition
```

---

**Example 2-56.**

Print a table with two columns, which associates students with their teachers. Only last names are printed.

```
get Teacher: instructor.LAST-NAME, Student-taught: student.LAST-NAME

where

exists enrl in COURSE-ENROLLMENT:

    enrl.THE-STUDENT = student and
    enrl.THE-OFFER. THE-INSTRUCTOR = instructor
```

When no heading for \( e_i \) is specified, then, by default, the following heading is assumed:

- If \( e_i \) ends in ".relation", then the heading is the relation.
- Otherwise the heading is the number \( i \).
Example 2-57.
The query
\[
\text{get } x.LAST-NAME, x.BIRTH-YEAR \\
\text{where } x \text{ is a STUDENT}
\]
produces a table with two columns, whose headings are:

\[LAST-NAME, BIRTH-YEAR\]

2.3.4. Integrity constraints in Logic

Specification of static integrity constraints — by assertions.

Example 2-58.
No student may be enrolled twice in the very same offering of a course.

for every enrl in COURSE-ENROLLMENT:

for every enrl2 in COURSE-ENROLLMENT:

if enrl.THE-STUDENT=\text{enrl2.THE-STUDENT} \text{ and}

enrl.THE-OFFERING=\text{enrl2.THE-OFFERING}

then enrl=\text{enrl2}

Specification of dynamic integrity constraints:

A *dynamic* integrity constraint is syntaxed as a static integrity constraint, but categories and relations may be suffixed with *-old* to denote the concepts of the previous state of the database.

Example 2-59.
The Student Council has secured that from 1991 once a grade has been reported (and thus, probably has been made known to the student) it may not be retroactively decreased.

for every enrl in COURSE-ENROLLMENT:

if \text{enrl.FINAL-GRADE-old>enrl.FINAL-GRADE}

then \text{enrl.THE-OFFERING.THE-QUARTER.THE-YEAR<1991}

Dynamic integrity constraints are rarely dictated by the logic of a user’s world, since their existence may cause irreversibility of some users’ errors.
Example 2-60.
In the previous example, a data-entry clerk who erroneously enters the grade of 80 instead of the correct grade of 70 may not be able to correct the typo.

2.3.5. *Extensions of Logic: aggregates, userviews, transactions, query forms

This is an advanced section.

2.3.5.1. *Aggregate operations: sum, count, average

Defined here is a second-order extension to enable set operations, such as summation, counting, etc. This is done by extending the syntax of expression with the summation quantifier:

$$\sum_{\text{variables}} \text{expression} \quad \text{where} \quad \text{condition}$$

Example 2-61.
The sum of the birth-years of all students =
$$\sum_{\text{s}} \text{s.BIRTH-YEAR} \quad \text{where} \quad \text{s is a STUDENT}$$

Example 2-62.
The number of pairs (instructor, department) where the instructor works in the department.

$$\sum_{\text{instructor,department}} \text{1} \quad \text{where} \quad \text{instructor WORKS-IN department}$$

(In the above formula, 1 is a constant function. Thus, for example, \(\sum_{i=1}^{5} 1\) equals 5. Adding up the constant 1 is thus the same as counting the object pairs satisfying the condition under the sum.)

The variables under \(\sum\) are quantified by the summation symbol. In addition to these variables, the condition and/or the expression may depend on other variables.

Rishe-DDS p. 2.27
Example 2-63.

The sum of the grades of student $s$. The sum depends on the variable $s$, meaning $s$ remains free in the sum.

$$\sum_{\text{enrl}} \text{enrl.FINAL-GRADE}$$

where

enrl is a COURSE-ENROLLMENT and not (enrl FINAL-GRADE null) and enrl THE-STUDENT's

Interpretation:

Let $e$ be an expression, and let $\phi(x_1, \ldots, x_n, y_1, \ldots, y_k)$ be a condition. Then the following is also an expression (it depends on the variables $y_1, \ldots, y_k$):

$$\sum_{x_1, \ldots, x_n} (e(x_1, \ldots, x_n, y_1, \ldots, y_k))$$

where

$\phi(x_1, \ldots, x_n, y_1, \ldots, y_k)$

When all the parameter-variables $y_1, \ldots, y_k$ are interpreted as some fixed objects, the sum yields a number. This number is the result of summation of the values of $e$ computed for every tuple of objects $x_1, \ldots, x_n$ satisfying $\phi(x_1, \ldots, x_n, y_1, \ldots, y_k)$.

The $\Sigma$ acts like a quantifier for $x_1, \ldots, x_n$. Therefore, although the subexpression $e$ does depend on $x_1, \ldots, x_n$, the whole $\Sigma$ expression does not. The variables $y_1, \ldots, y_n$ remain unquantified.

Alternative (linear) notation (we would not use the two-dimensional notation of $\Sigma$ in a real computer language):

$$\sum e$$

for $x_1, \ldots, x_n$

where $\phi(x_1, \ldots, x_n, y_1, \ldots, y_k)$

Abbreviation:

When $x_1, \ldots, x_n$ are exactly the variables on which the expression $e$ depends (that is, all $x_i$ and none of $y_i$ appear free in the expression $e$), the for clause may be omitted:

$$\sum e \text{ where } \phi(x_1, \ldots, x_n, y_1, \ldots, y_k)$$

Example 2-64.

For every information systems student, print his last name and the sum of his grades.

get

student.LAST-NAME,
(sum enrollment.FINAL-GRADE
   where enrollment THE-STUDENT student)
   where
   student is a STUDENT and
   (student.MAJOR DEPARTMENT-NAME ‘Information Systems’)

Example 2-65.
Print the sum of all the grades given by Prof. Smith. This query outputs only one value (the sum). There is no ‘where condition’ for the ‘get’ of the query.

get (sum enrollment.FINAL-GRADE
   where
   ‘Smith’=enrollment.THE-OFFERING.
   THE-INSTRUCTOR.LAST-NAME)

Abbreviation for count:

count x₁, . . . , xₙ where φ(x₁, . . . , xₙ, y₁, . . . , yₖ)

stands for:

sum 1 for x₁, . . . , xₙ where φ(x₁, . . . , xₙ, y₁, . . . , yₖ).

Example 2-66.
How many students are there in the university?

get (count std where std is a STUDENT)

Abbreviation for average:

average e ...

stands for:

(sum e ...) / (count e ...)

Example 2-67.
What students have their average grade below 60?

get std.LAST-NAME
   where std is a STUDENT and
   60 > (average enrl.FINAL-GRADE
   where enrl THE- STUDENT std)

Note: this query could not be formulated as follows, since only the distinct grades would be then taken into account:

Rishe-DDS p. 2.29
get std.LAST-NAME
where 60 >
    average fgrade
where
    exists enrl in ENROLLMENT:
        (enrl FINAL-GRADE fgrade and
         enrl THE-Student std)

If a student has three enrollments with grades 100, 50, and 100, then the average calculated in the first version would be 250/3=83 (correct), and in the second version 150/2=75 (incorrect).

2.3.5.2.  *Shorthand notation for n-ary relationships

Example 2-68.
Often we need to specify a condition like:

The instructor i offered course c in quarter q.

In calculus this can be stated as:

exists offer in COURSE-OFFERING:
    offer THE-INSTRUCTOR i and
    offer THE-COURSE c and
    offer THE-QUARTER q

The above statement can be written in a shorthand notation as:

COURSE-OFFERING
    (THE-INSTRUCTOR: i,
    THE-COURSE: c,
    THE-QUARTER: q)

Abbreviation:

category (relation_1: expression_1, . . . , relation_k: expression_k)
stands for

exists x in category:
    (x relation_1 expression_1 and · · · and x relation_k expression_k)

Example 2-69.
Print the names of the courses taught by Prof. McFarland.
2.3.5.3. *Inference rules of userviews

Reminder:

A userview consists of an alternative schema and inference rules. The inference rules specify the categories and relations of the alternative schema in terms of the categories and relations of the original schema.

**Specification of an inferred relation**

\[
\text{userview relation}: \ expression_1 \ new-relation \ expression_2 \\
\text{where condition}
\]

*Example 2-70.*

The following is a specification of an inferred relation \textit{TAUGHT} (between instructors and students) in terms of the relations existing in the schema.

\[
\text{userview relation}: \text{instructor} \ \text{TAUGHT} \ \text{student} \\
\text{where}
\]

\[
\exists \text{enrl in COURSE-ENROLLMENT}: \\
\text{enrl.THE-STUDENT} = \text{student and} \\
\text{enrl.THE-OFFER. THE-INSTRUCTOR} = \text{instructor}
\]

*Example 2-71.*

The following is a specification of an inferred relation

- \textit{average-grade} –attribute of \textit{STUDENT}, range: 0..100 (\textit{m:1})

\[
\text{userview relation}: \text{student} \ \text{AVERAGE-GRADE} \\
(\text{average e.FINAL-GRADE} \\
\text{where} \\
\ e \text{ is a COURSE-ENROLLMENT and}
\]

Rische-DDS p. 2.31
Specification of an inferred abstract category

Usually, but not always, an inferred abstract category is a new subcategory of an existing category. Its purpose is usually to restrict the userview user to a subset of objects which are relevant to that user’s task. An inferred category can be specified as a set of objects which is derived from the categories and relations of the database. The specification is:

\[ \text{userview subcategory: expression is a new-subcategory} \]
\[ \text{where condition} \]

**Example 2-72.**

userview subcategory: \( s \) is a \( \text{COMPUTER-SCIENCE-MAJOR} \)
where

\( s \) is a \( \text{STUDENT} \) and
\( s.MAJOR \text{ NAME} \) ‘Computer Science’

Specification of representation of relationships by a category

The definition is preceded by an example.

**Example 2-73.**

We wish to have (in the userview) a category of events of work of instructors in departments:

- \( \text{WORK} \rightarrow \text{category} \)
- \( \text{the-department} \rightarrow \text{relation from WORK to DEPARTMENT (m:1)} \)
- \( \text{the-instructor} \rightarrow \text{relation from WORK to INSTRUCTOR (m:1)} \)

The user of the user-view will perceive the category \( \text{WORK} \) as containing objects (events) which are distinct from any other objects in the database. These new objects will be perceived only through the userview, without actually existing in the database. Thus, they are \( \text{virtual} \) objects.

This is done by specifying a category of virtual objects with relationships to existing objects. The syntax is:

userview category:
\[ \text{new-category (relation_1: expression_1, \ldots, relation_k: expression_k)} \]
\[ \text{where condition} \]
Let $x_1, \ldots, x_n$ be the variables on which the condition and the expressions depend. (They must depend on the same variables.) For every tuple of values of variables satisfying the condition, one new virtual object is created. That object is related by the new relations $relation_1, \ldots, relation_k$ to the values of the expressions $expression_1, \ldots, expression_k$.

**Example 2-74.**

The following is a specification of an inferred category `WORK` and two inferred relations, `THE-DEPARTMENT` and `THE-INSTRUCTOR`, whose domain is `WORK`. The specification is in terms of the the relations existing in the schema.

**userview category:**

`WORK (THE-DEPARTMENT: department, THE-INSTRUCTOR: instructor)`

**where** instructor WORKS-IN department

### 2.3.5.4. *Transactions*

This section extends Predicate Calculus to allow specification of transactions — creation of sets of objects, categorization and decategorization of objects, relating and unrelating objects, and so on.

The operations in Calculus are usually not atomic but work on sets of objects. One single operation can create a set of new objects, place them in categories, and relate them to different existing objects by several relations.

**Creation** of new abstract objects and relating them to existing or concrete objects:

```
insert into category
(relation_1 : expression_1, \ldots, relation_k : expression_k)
where condition
```

- If no where clause is specified, then only one new abstract object is created. This object is put into the category and related by the relations to the values of the expressions.

**Example 2-75.**

Create a new department named ‘Computer Engineering’

```
insert into DEPARTMENT (NAME: ‘Computer Engineering’)
```

- Some of the names of the relations may be identical. This allows one object to be related to several objects by one relation (m:m or 1:m).
Example 2-76.
Create a new department named ‘Computer Engineering’ and ‘CE’.

```
insert into DEPARTMENT (NAME: ‘Computer Engineering’, NAME: ‘CE’)
```

- If a `where` clause is specified with a `condition` on variables $x_1, \ldots, x_n$, then for every tuple of values of the variables satisfying the `condition`, one new object is created and related accordingly.

Example 2-77.
Enroll the computer science student Jack Johnson into the Databases course given by Prof. Smith in Fall 1990.

```
insert into COURSE-ENROLLMENT (THE-STUDENT: s, THE-OFFERING: offer) where
  s.MAJOR NAME = ‘Computer Science’ and
  s.LAST-NAME = ‘Johnson’ and
  s.FIRST-NAME = ‘Jack’ and
  offer.THE-QUARTER.YEAR = 1990 and
  offer.THE-QUARTER.SEASON = ‘Fall’ and
  offer.THE-COURSE.NAME = ‘Databases’ and
  offer.THE-INSTRUCTOR.LAST-NAME = ‘Smith’
```

- The variables on which the `condition` depends must be those on which the `expressions` depend.

Example 2-78.
Enroll all computer science students into the Databases course given by Prof. Smith in Fall 1990.

```
insert into COURSE-ENROLLMENT (THE-STUDENT: s, THE-OFFERING: offer) where
  s.MAJOR NAME = ‘Computer Science’ and
  offer.THE-QUARTER.YEAR = 1990 and
  offer.THE-QUARTER.SEASON = ‘Fall’ and
  offer.THE-COURSE.NAME = ‘Databases’ and
```
Rishe-DDS p. 2.34
- When the insert statement calls for an insertion of a new object while there is already an object having the same relationships as those of the new object, the new object is not inserted.

Example 2-79.
If the department named ‘Management’ already exists, then the following command produces no effect:

```
insert into DEPARTMENT (NAME: ‘Management’)
```

**Connection** between existing abstract objects, between existing abstract objects and concrete objects, between existing abstract objects and categories:

```
connect fact_1, ... , fact_k [where condition ]
```

Each fact_i is either

```
expression_i category_i
```

or

```
expression_i relation_i expression_i
```

**Interpretation:**
- If no where clause is specified, the values of the expressions are related by the relations and/or categorized by the categories.
- If a where clause is specified with a condition \( \phi \) on variables \( x_1, \ldots, x_n \), then for every tuple of values of the variables satisfying \( \phi \) the values of the expressions are related and categorized as above.
- The variables on which the condition \( \phi \) depends must be those on which the expressions depend.

Example 2-80.
Let ‘CS’ be an alternative name for the department named ‘Computer Science’.

```
connect dept NAME ‘CS’
where
depth is a DEPARTMENT and
depth NAME ‘Computer Science’
```
Example 2-81.
Give the grade 100 to the computer science student Jack Johnson enrolled in the Databases course given by Prof. Smith in Fall 1990.

```
connect enrl FINAL-GRADE 100
where
  enrl.THE-STUDENT.MAJOR NAME = 'Computer Science' and
  enrl.THE-STUDENT.LAST-NAME = 'Johnson' and
  enrl.THE-STUDENT.FIRST-NAME = 'Jack' and
  enrl.THE-OFFERING.THE-QUARTER.YEAR = 1990 and
  enrl.THE-OFFERING.THE-QUARTER.SEASON = 'Fall' and
  enrl.THE-OFFERING.THE-COURSE.NAME = 'Databases' and
  enrl.THE-OFFERING.THE-INSTRUCTOR.LAST-NAME = 'Smith'
```

Example 2-82.
Give the grade 100 to all computer science students enrolled in the Databases course given by Prof. Smith in Fall 1990.

```
connect enrl FINAL-GRADE 100
where
  enrl.THE-STUDENT.MAJOR NAME = 'Computer Science' and
  enrl.THE-OFFERING.THE-QUARTER.YEAR = 1990 and
  enrl.THE-OFFERING.THE-QUARTER.SEASON = 'Fall' and
  enrl.THE-OFFERING.THE-COURSE.NAME = 'Databases' and
  enrl.THE-OFFERING.THE-INSTRUCTOR.LAST-NAME = 'Smith'
```

**Removal of connections and removal of objects:**

```
disconnect fact_1, . . . , fact_k [where condition]
```

**Interpretation:**

- If no `where` clause is specified, the values of the expressions are unrelated and/or decategorized. Objects that are removed from all their categories are removed from the database.
- If a `where` clause is specified with a condition \( \phi \) on variables \( x_1, \ldots, x_n \), then for every tuple of values of the variables satisfying \( \phi \) the values of the expressions are unrelated and decategorized as above.
• The variables on which the condition depends must be those on which the expressions of the facts depend.

**Example 2-83.**
Let ‘CS’ no longer be an alternative name of a department.

```sql
disconnect dept NAME 'CS'
where
department is a DEPARTMENT and dept NAME 'CS'
```

**Example 2-84.**
The computer science student Jack Johnson has dropped the *Databases* course given by Prof. Smith in Fall 1990.

```sql
disconnect enr ENROLLMENT
where
enrollment.THE-STUDENT.MAJOR NAME = 'Computer Science' and
enrollment.THE-STUDENT.LAST-NAME = 'Johnson' and
enrollment.THE-STUDENT.FIRST-NAME = 'Jack' and
enrollment.THE-OFFERING.THE-QUARTER.YEAR = 1990 and
enrollment.THE-OFFERING.THE-QUARTER.SEASON = 'Fall' and
enrollment.THE-OFFERING.THE-COURSE.NAME = 'Databases' and
enrollment.THE-OFFERING.THE-INSTRUCTOR.LAST-NAME = 'Smith'
```

**Example 2-85.**
Void all the grades in the *Databases* course given by Prof. Smith in Fall 1990.

```sql
disconnect enr FINAL-GRADE enr.FINAL-GRADE
where
enrollment.THE-OFFERING.THE-QUARTER.YEAR = 1990 and
enrollment.THE-OFFERING.THE-QUARTER.SEASON = 'Fall' and
enrollment.THE-OFFERING.THE-COURSE.NAME = 'Databases' and
enrollment.THE-OFFERING.THE-INSTRUCTOR.LAST-NAME = 'Smith'
```

**Correction of facts:**

Rishe-DDS p. 2.37
**update** fact₁, . . . , factₖ [where condition]

This is a combination of *disconnect* and *connect*. Before a connection \( aRb \) is made, the relationships \( aRx \) are removed for every \( x \). Before a connection \( aC' \) is made, the facts \( aC \) are removed for every \( C \).

**Example 2-86.**

Let ‘CS’ be the new name instead of ‘Computer Science’.

**update** dept NAME ‘CS’
**where**

dept **is a** DEPARTMENT **and**

department NAME ‘Computer Science’

**Example 2-87.**

Give the grade 100 to the computer science student Jack Johnson enrolled in the Databases course given by Prof. Smith in Fall 1990. If a grade has been previously given, replace it by the new grade.

**update** enrl FINAL-GRADE 100
**where**

enrl.**THE-STUDENT.MAJOR NAME** ‘Computer Science’ **and**
enrl.**THE-STUDENT.LAST-NAME**=‘Johnson’ **and**
enrl.**THE-STUDENT.FIRST-NAME**=‘Jack’ **and**
enrl.**THE-OFFERING.THE-QUARTER.YEAR**=1990 **and**
enrl.**THE-OFFERING.THE-QUARTER.SEASON**=‘Fall’ **and**
enrl.**THE-OFFERING.THE-COURSE.NAME**=‘Databases’ **and**
enrl.**THE-OFFERING.THE-INSTRUCTOR.LAST-NAME**=‘Smith’

**Example 2-88.**

Increase by 10 percent the grades of all computer science students enrolled in the Databases course given by Prof. Smith in Fall 1990.

**update** enrl **FINAL-GRADE** 1.1×enrl.**FINAL-GRADE**
**where**

enrl.**THE-STUDENT.MAJOR NAME** ‘Computer Science’ **and**
enrl.**THE-OFFERING.THE-QUARTER.YEAR**=1990 **and**
enrl.**THE-OFFERING.THE-QUARTER.SEASON**=‘Fall’ **and**

Rishe-DDS p. 2.38
2.3.5.5. **Parametric query forms**

Often the users ask similar queries which differ only in the values of some parameters.

*Example 2-89.*

1. What are the grades of the student whose name is ‘Jackson’?
2. What are the grades of the student whose name is ‘Smith’?

It is desirable that such queries are predefined in parametric form, and the users would supply only the values of the parameters.

*Example 2-90.*

What are the grades of the student whose name is $x$, where $x$ is supplied by the end user when the query runs?

Such a predefinition is called a **query in parametric form** or **query form**. It saves time on specification of similar queries and allows the less-sophisticated end users to use queries which can be specified only by more sophisticated users, such as programmers and analysts.

In calculus, query forms are specified by the following syntax:

- **depending on** parameters
- **get** expressions
- **where** condition

The condition and the expressions may depend on the parameters.

*Example 2-91.*

What are the grades of the student whose name is $x$, where $x$ is supplied by the end user when the query runs?

- **depending on** $x$
- **get** $e$.THE-OFFERING.THE-COURSE.NAME, $e$.FINAL GRADE
- **where**
  - $e$ is an ENUMRLMENT and
  - $e$.THE-STUDENT.LAST-NAME = $x$
2.3.6. *Nontotal functional relations: interpretation of the dot-application*

If $f$ is not total, then $e.f$ may be ambiguous. However, a smart DBMS may be able to follow the user’s intuition in using such expressions. In order to provide a meaningful result, the dot-application $e.f$ of a nontotal functional relation $f$ to an expression $e$ is interpreted by the smart DBMS by analyzing the whole condition or assertion containing the dot-application.

**Example 2-92.**

Consider the following assertion which contains a dot-application of the nontotal relation $BIRTH\text{-}YEAR$.

\[
\text{for every } y \text{ in STUDENT:} \\
y \cdot BIRTH\text{-}YEAR > 1980
\]

This assertion will be interpreted by a smart DBMS as

\[
\text{for every } y \text{ in STUDENT:} \\
\text{exists } x \text{ in Integer:} \\
y \cdot BIRTH\text{-}YEAR \times \text{ and } x > 1980
\]

This interpretation of the dot-application of nontotal functional relations can be defined formally as follows.

An expression $e.f$, where $e$ is an expression and $f$ is a database functional relation, is regarded as a syntactic abbreviation. Let $x_1, \ldots, x_k$ be the variables on which the expression $e$ depends. For the above example, the only such variable is $y$.

Let $\phi$ be the largest subexpression (within the whole assertion or condition) containing $e.f$ and still depending on all the variables $x_1, \ldots, x_k$; that is, none of these variables is quantified in the subformula $\phi$. ($\phi$ may depend also on additional variables.) For the above example,

\[
\phi = (y \cdot BIRTH\text{-}YEAR > 1980)
\]

Let $C$ be the range of $f$.

Let $\psi = \phi|_{e.f}$. (That is, $\psi$ is obtained from $\phi$ by substitution of a new variable $x$ for all the occurrences of $(e.f)$ in $\psi$.) For the above example,

\[
\psi = (x > 1980)
\]

Then $\phi$ stands for:

\[
(\text{exists } x \text{ in } C : ((e \cdot f \cdot x) \text{ and } \psi))
\]
Reference Schemas

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.

Figure Ref-1. A semantic schema for a university application.
Figure Ref-2. A relational schema for the university application.
3. FROM THE SEMANTIC TO THE RELATIONAL MODEL

Chapter 3 defines the Relational Data Model and presents a top-down methodology for design of relational databases. The theoretical foundations of the relational model were introduced by E. Codd in 1970. By late 1980s this model had become the state of the art in commercial database management.

3.1. Time Invariant Attributes and Keys

Time-invariant attribute — An attribute $A$ is time-invariant if once an object $x$ becomes related by $A$ to a value $y$, the object $x$ will forever be related by $A$ to $y$, as long as $x$ exists.

There are no time-invariant attributes in the natural user world. Even if the laws of physics or society do not allow for an attribute to change in time, the attribute may change in the perceived real world due to discoveries of errors in earlier perception. For example, a social security number could be wrongly reported and then corrected. Thus, time-invariance is defined only in implementational restrictions. Such restrictions are unavoidable in the relational database design. The methodology of relational schema design that is presented below has among its goals the minimization of the negative effect of such implementational restrictions.

Example 3-1.

None of the attributes given in the previous examples is truly time-invariant. The following attributes are the next closest thing: they change only when an error is discovered.

- birth-year of PERSON;
- year and season of QUARTER.

The following attributes may sometimes change in time ‘‘in the real world.’’ Thus, declaring them as time-invariant would be a stronger implementational restriction.

- last-name and first-name of PERSON;
- name of COURSE.

The following attributes have a high probability of change in time; no reasonable database design would restrict them as time-invariant.

- address of PERSON;
- final-grade of ENROLLMENT.

Keys

Rishe-DDS p. 3.1
1. **Single-attribute key**

A time-invariant attribute of a category is called its *key* if it is 1:1 and total. That means that the values of the attribute can be used to identify the objects of the category.

**Example 3-2.**

If we assume that the attribute *name* of *COURSE* is time-invariant, then it is probably also a key since:

- (a) It is total, provided we do not want our database to contain courses without names
- (b) It is 1:1, that is, no two courses have the same name, and no course has two names.

Due to the *time-invariance* requirement, no attribute is really a key in the natural user’s world. Thus, the property of a *key* is defined only in implementational restrictions, which are unavoidable in the relational database design. Also, the requirement of *totality* is very rarely an integrity constraint imposed by the logic of the user world but rather is an implementational restriction.

**Example 3-3.**

Would the attribute *social-security-number* of the category *PERSON* be its key? Let us assume that we have already imposed an implementational restriction of *time-invariance* on this attribute (thus making it very hard and dangerous for our users to correct errors).

To be a key, this attribute must be one-to-one and total. It is indeed one-to-one. But in the real world it is not really total: there are some persons who do not have social security numbers before they are reported to our database. The totality can be imposed as an implementational restriction with one of the following practical provisions:

- (a) Persons who do not have social security numbers would be assigned the dummy default number 0, which would be called a “*social security number*” for the purposes of the database. But then this attribute would no longer be one-to-one since two persons may have the same number 0.

- (b) For persons who do not have real social security numbers, generate dummy temporary numbers in such a way that all these numbers are different and not in the range of possible real social security numbers. For example, if the real s.s. numbers may never begin with the digit 9, then begin all the dummy numbers with this digit.

Apart from the unnaturalness and “*cheating*” which are bound to result in misinterpretation of the computer’s reports, there is a serious technical problem: What if a person did not have a real s.s. number at first, but later received one, and this new number is a valuable piece of...
information which the user wants to keep in the database? If we allow replacement of the old dummy number by the new real one, then the time-invariance requirement is violated.

c. Yet another possibility is to disallow recording of information about persons who do not have s.s. numbers. I am afraid that our client might be unhappy with such a restriction.

**Example 3-4.**

We can generate a new artificial attribute to serve as a key. Let id# be a new artificial attribute of the category PERSON, generated in such a way that when new persons enter the database, they are assigned arbitrary meaningless numbers which have not been used before. (When a person is deleted from the database, the number is not reused.) This attribute would be a key.

**Convention:** In this text, we shall name the attributes constrained to be keys with the suffix -key.

**Example 3-5.**

\textit{name-key} of \textit{COURSE}

The name of the attribute \textit{name-key} implicitly defines a constraint or implementational restriction "This attribute is a key of its domain, the category COURSE."

2. **Multiattribute key**

The following definition extends the concept \textit{key} to a collection of attributes.

**Key** of a category — a \textit{collection} of total time-invariant attributes \( f_1, f_2, \ldots, f_n \) whose domain is that category and which satisfy two requirements:

(i) For any collection of values, \( x_1, \ldots, x_n \) there is no more than one object \( y \) of the category s.t.

\[
x_1 = y.f_1 \quad \text{and} \quad x_2 = y.f_2 \quad \text{and} \quad \cdots \quad \text{and} \quad x_n = y.f_n
\]

(ii) No proper subcollection of these attributes \textit{always} satisfies (i).

Practically, requirement (i) means that the collection of attributes is sufficient to identify every object of the category. Requirement (ii) means that the collection is minimal: if one of the attributes is not known then the remaining attributes might not provide sufficient information to identify every object of the category.

Rishe-DDS p. 3.3
Example 3-6.
*(the-year, the-season)* is the only key of the category *QUARTER*.
Either attribute alone does not identify each object. Together they do.

**Convention:** In this text, when a category is constrained to have exactly one key, and the key is composed of several attributes, we shall name these attributes with the suffix *-in-key*.

Example 3-7.
*the-year-in-key* and *the-season-in-key* of *QUARTER*

**Note:**

(i) In the real world a category usually has *no* key. Thus, the existence of a key is usually not an integrity constraint but rather an implementational restriction. This restriction will be imposed when unavoidable due to limitations of a DBMS or a database model, especially the relational model.

(ii) Existence of a key makes every object of the category identifiable with the values of the key and eliminates the necessity to refer to abstract objects.

Example 3-8.
Courses can be identified with strings *name-key*. Quarters can be identified with pairs *(the-year-in-key, the-season-in-key)*.

(iii) A category which has no key may still have all its objects completely identifiable (using different relations and their combinations for different objects), but the identification would not be uniform.

Example 3-9.
The category *DEPARTMENT* does not have a key (it does not have any attribute at all; the relation *NAME* is not an attribute because it is 1:m). However, if the relation *department-name* is total, then, being 1:m, it distinguishes between every two departments according to their names.

If the relation *department-name* is not total, then we might have a problem distinguishing between two departments that have no names at all. In that case, it is possible that the "anonymous" departments can be distinguished by their relationships of works-in with instructors.
Example 3-10.

We have not defined any key for the category INSTRUCTOR. However, most instructors have unique names and can be identified by their names. Few instructors have common names, and their identification requires names of their departments in addition to their first and last names. Some of the instructors have nonunique names but do not work for any department at all. These rare persons can be identifiable by their names in conjunction with their birth years or with courses they teach or with their addresses (whichever of this information is available and sufficient for identification).

This identification is nonuniform because different instructors are identified by different relations.

The nonuniform identification is quite common and satisfactory in the real world and in the Binary Model, but the implementational restrictions of the Relational Model will necessitate a uniform identification of the objects of a category.

(iv) When a key is composed of several attributes it is still one key.

(v) A category may theoretically have several keys. However, since categories in the real world rarely have even one key, the existence of more than one key would be an unnecessarily strong implementational restriction, which is not required by database management systems. Thus, the possibility of multiple keys will be ignored in this text.

3.2. Relational Schemas Defined

A binary schema is called table-oriented or a relational schema if:

(i) All the abstract categories of the schema have keys.

(ii) All the abstract categories are pairwise disjoint.

(iii) The only relations are attributes.

Thus, all the information in a relational schema is represented by attributes of categories.

Example 3-11.

Figure ___ (on the last page of this book) could be a relational schema for the university application, provided:

- All the categories are restricted to having keys as shown.
- There are no persons but students and instructors.
- The categories INSTRUCTOR and STUDENT are disjoint.
We shall see later that this schema can be used even without imposing the severe restriction of disjointness.

Example 3-12.
Consider the category COURSE-ENROLLMENT. In the relational schema it has no relations to the categories STUDENT and COURSE-OFFERING. Instead, it has attributes giving essentially the same information. Thus, instead:

- the-student — relation from COURSE-ENROLLMENT to STUDENT (m:1)

the category has the attribute:

- student-id-in-key — attribute of COURSE-ENROLLMENT, range: Integer (m:1)

Table-declaration, relation-declaration — any subschema of a relational schema having only one abstract category with all of its attributes and their ranges. The name of the table is the name of that category.

Example 3-13.
Here is the table-declaration QUARTER.

QUARTER

year-in-key: 1980..1995
season-in-key: String

Instantaneous table, instantaneous relation — an instantaneous database viewed under a table-declaration. This means that an instantaneous table is a part of an instantaneous relational database containing all the objects of one table and all their relationships (attributes).

Representation of an instantaneous table — a printable table whose title is the name of the category, the names of the columns are the attributes, and for every object in the category there is a row (called a tuple) composed of the values of the attributes of that object.
Example 3-14.
The following is an instantaneous table of STUDENT. (Of course, in this example the instantaneous table is unrealistically small. It would contain thousands of tuples for a normal university.)

\[
\begin{array}{|l|l|l|l|l|l|l|}
\hline
\text{id-key} & \text{last-name} & \text{first-name} & \text{birth-year} & \text{address} & \text{major-dept\-main-name} & \text{minor-dept\-main-name} \\
\hline
12345 & Victory & Elizabeth & 1966 & 100 Sun St. & Computer Science & Economics \\
12348 & Howards & Jane & 1965 & 200 Dorms & Arts & Economics \\
43532 & Wood & Michael & 1964 & 110 Dorms & Arts & Economics \\
\hline
\end{array}
\]

\textbf{Figure 3-1.} An instantaneous table.

\textit{Note:}

(i) There cannot be two identical rows in an instantaneous table because this would imply that there are two objects with the same values of the key.

(ii) The order of tuples (rows) is immaterial; the order of columns is immaterial.

Example 3-15.
The following figure represents the very same instantaneous table as the previous example.

\[
\begin{array}{|l|l|l|l|l|l|l|}
\hline
\text{birth-year} & \text{address} & \text{major-dept\-main-name} & \text{id-key} & \text{last-name} & \text{first-name} & \text{minor-dept\-main-name} \\
\hline
1964 & 110 Dorms & Arts & 43532 & Wood & Michael & Economics \\
1965 & 200 Dorms & Arts & 12348 & Howards & Jane & Economics \\
1966 & 100 Sun St. & Computer Science & 12345 & Victory & Elizabeth & Economics \\
\hline
\end{array}
\]

\textbf{Figure 3-2.} Some columns and rows have been moved in the representation without changing the instantaneous table.

\textbf{Representation of an instantaneous relational database} — a collection of instantaneous tables.
Example 3-16.
The following figure is a representation of an instantaneous database for the schema of Figure ___. This instantaneous database represents the same state of the application’s real world as the binary instantaneous database of Figure ___ on page ___.

<table>
<thead>
<tr>
<th>student</th>
<th>id-key</th>
<th>last-name</th>
<th>first-name</th>
<th>birth-year</th>
<th>address</th>
<th>major-dept-main-name</th>
<th>minor-dept-main-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>Victory</td>
<td>Elizabeth</td>
<td>1966</td>
<td>100 Sun St.</td>
<td>Computer Science</td>
<td></td>
<td>Economics</td>
</tr>
<tr>
<td>12348</td>
<td>Howards</td>
<td>Jane</td>
<td>1965</td>
<td>200 Dorms</td>
<td>Arts</td>
<td></td>
<td>Economics</td>
</tr>
<tr>
<td>43532</td>
<td>Wood</td>
<td>Michael</td>
<td>1964</td>
<td>110 Dorms</td>
<td>Arts</td>
<td></td>
<td>Economics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instructor</th>
<th>id-key</th>
<th>last-name</th>
<th>first-name</th>
<th>birth-year</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>Brown</td>
<td>George</td>
<td>1956</td>
<td>112 Lucky Dr.</td>
<td></td>
</tr>
<tr>
<td>14352</td>
<td>Whatson</td>
<td>Mary</td>
<td>1953</td>
<td>231 Fortune Dr.</td>
<td></td>
</tr>
<tr>
<td>24453</td>
<td>Blue</td>
<td>John</td>
<td>1950</td>
<td>536 Orange Dr.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>department</th>
<th>main-name-key</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computer Science</td>
</tr>
<tr>
<td></td>
<td>Mathematics</td>
</tr>
<tr>
<td></td>
<td>Physics</td>
</tr>
<tr>
<td></td>
<td>Arts</td>
</tr>
<tr>
<td></td>
<td>Economics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>department naming</th>
<th>name-key</th>
<th>main name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CS</td>
<td>Computer Science</td>
</tr>
<tr>
<td></td>
<td>Math</td>
<td>Mathematics</td>
</tr>
<tr>
<td></td>
<td>Physics</td>
<td>Physics</td>
</tr>
<tr>
<td></td>
<td>Mathematics</td>
<td>Mathematics</td>
</tr>
<tr>
<td></td>
<td>Computer Science</td>
<td>Computer Science</td>
</tr>
<tr>
<td></td>
<td>Arts</td>
<td>Arts</td>
</tr>
<tr>
<td></td>
<td>Economics</td>
<td>Economics</td>
</tr>
</tbody>
</table>

Figure 3-3. An instantaneous database for the relational schema of the university application.
WORK

<table>
<thead>
<tr>
<th>instructor-id-in-key</th>
<th>department-main-name-in-key</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>Computer Science</td>
</tr>
<tr>
<td>11332</td>
<td>Mathematics</td>
</tr>
<tr>
<td>14352</td>
<td>Physics</td>
</tr>
<tr>
<td>24453</td>
<td>Mathematics</td>
</tr>
</tbody>
</table>

COURSE

<table>
<thead>
<tr>
<th>name-key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Databases</td>
</tr>
<tr>
<td>Football</td>
</tr>
<tr>
<td>Gastronomy</td>
</tr>
</tbody>
</table>

QUARTER

<table>
<thead>
<tr>
<th>year-in-key</th>
<th>season-in-key</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>Fall</td>
</tr>
<tr>
<td>1990</td>
<td>Winter</td>
</tr>
<tr>
<td>1990</td>
<td>Spring</td>
</tr>
</tbody>
</table>

COURSE OFFERING

<table>
<thead>
<tr>
<th>instructor-id-in-key</th>
<th>course-name-in-key</th>
<th>year-in-key</th>
<th>season-in-key</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>Databases</td>
<td>1990</td>
<td>Fall</td>
</tr>
<tr>
<td>11332</td>
<td>Football</td>
<td>1990</td>
<td>Fall</td>
</tr>
<tr>
<td>11332</td>
<td>Gastronomy</td>
<td>1990</td>
<td>Fall</td>
</tr>
</tbody>
</table>

COURSE ENROLLMENT

<table>
<thead>
<tr>
<th>instructor-id-in-key</th>
<th>course-name-in-key</th>
<th>year-in-key</th>
<th>season-in-key</th>
<th>student-id-in-key</th>
<th>final-grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>Gastronomy</td>
<td>1990</td>
<td>Fall</td>
<td>12345</td>
<td>100</td>
</tr>
<tr>
<td>11332</td>
<td>Gastronomy</td>
<td>1990</td>
<td>Fall</td>
<td>12348</td>
<td>70</td>
</tr>
<tr>
<td>11332</td>
<td>Databases</td>
<td>1990</td>
<td>Fall</td>
<td>12348</td>
<td>80</td>
</tr>
</tbody>
</table>

Figure 3-3. Continued.

An alternative representation of relational schemas (linear, nongraphic):

```
  table name [ attribute : range, . . . , attribute : range ]

  . . .

  table name [ attribute : range, . . . , attribute : range ]
```

Example 3-17.

Here is a linear representation of the schema of Figure ___:

Risque-DDS p. 3.9
3.3. Implementational Restrictions: Pros and Cons

Consequences of implementational restrictions in the Relational Model:

- The schemas often deviate from the real world.
- The schemas are often unnatural, inflexible, and redundant.
- The integrity constraints are often underrepresented in the schemas.
- The queries are usually harder to specify.

Example 3-18.
Instead of

\[(i \ WORKS-IN \ d)\]

we have to say:

\[(\text{exists} \ w \ \text{in} \ WORK:\]

\[i.\text{ID-key} = w.\text{INSTRUCTOR-ID-in-key} \ \text{and} \]

\[d.\text{MAIN-NAME-key} = w.\text{DEPARTMENT-MAIN-NAME-in-key})\]

Purposes of implementational restrictions in the Relational Model:

- Exclusion of nonattribute relations,
- exclusion of intersecting categories,
- exclusion of sub- and supercategories — allow for readable and simple
representation of the instantaneous database as everyday tables.

- **Totality and uniqueness of keys** — allow:
  - A standard printable representation for every object
  - Readable reference to objects of one table from objects of another table
  - Unambiguous definition of simple updates

- **Time-invariance of keys** — prevents inconsistent update of keys.

  If the values of the key of an object are updated, then all the references to this object throughout the instantaneous database become wrong. The human user cannot be relied upon to find and update all the references. The database management system is normally unaware of these references and thus cannot update them automatically, unless it is a very advanced system having a high-level support for the so-called referential integrity.

  Many database management systems do not explicitly require the time-invariance but do not provide a high-level support for the referential integrity either. This does not mean that those systems do not need the time-invariance. Rather, this means that they do not check for the time-invariance, and, without warning to the user, they corrupt the database when the user modifies a value of a key.

  Note: It is hard to say whether the severe implementational restrictions can really be justified by the benefits of the model.

**Totality of nonkey attributes**

Many relational database management systems require a further implementational restriction: they require that all the attributes be total.

Relational DBMS that do not require the totality of attributes allow null values of attributes.

The restriction of the totality of the nonkey attributes pragmatically requires a modification of the meaning of an attribute which is nontotal in the real world. A special value is identified which can never be a value of the attribute in the real world. This value is assigned to the objects that do not have any real value of the attribute.

---

**Example 3-19.**

We can assign the dummy grade "-1" to the enrollments which do not have any real grade. Thus we convert:

- `final-grade` — attribute of `COURSE-ENROLLMENT`, range: `0..100` `(m:1)`

into:

- `new-final-grade` — attribute of `COURSE-ENROLLMENT`, range: `-1..100` `(m:1)`

so that:

Rishe-DDS p. 3.11
Such “‘cheating’” will, however, cause inconvenience and misinterpretation of queries of naive users.

\[
\begin{align*}
x \cdot NEW\text{-FINAL-GRADE} &= -1 \text{ iff } \\nonumber \\
x \cdot FINAL\text{-GRADE} &= null
\end{align*}
\]

Example 3-20.
The query
\[
\text{get enrl.STUDENT-ID where enrl is a COURSE-ENROLLMENT and enrl.new-FINAL-GRADE < 60}
\]
will also retrieve the students who have no grades at all. The system will also believe that two students who really have no grades have identical grades, and thus will mislead the user.

Some relational database management systems allow the analyst to define default values of attributes. These would be the values for the objects when the user has entered no value. The definition of default values enhances the convenience of updates, but it does not solve the problem of the misinterpretation of queries. A default value is still a regular value as far as the system is concerned.

Example 3-21.
We can define "-1" to be the default grade. This will be the grade of all the enrollments for which the user has not provided a grade.

In many cases the default value is defined as 0 or the empty string.

Example 3-22.
We should not define 0 as the default for the grade if our intention is to use the default value as a substitute for null. This is because 0 might be a real grade of a poor student.

We can use the empty string ‘’ as the default for last-name.
3.4. Relational Database Design

The purpose of this section is to show how a high-quality relational database can be designed once we have a semantic binary schema for the application.

3.4.1. Design principles

Schema-conversion — replacement of a schema by another schema having the same information content. This means that each of the two schemas can be regarded as a userview of the other.

Schema-conversion is a means of database design: a schema is first designed in a higher-level database model and then translated into a lower-level model which is supported by the available DBMS (when a DBMS for a higher-level model is unavailable or inadequate).

Note:

(i) Schema conversion is usually done in order to impose implementation restrictions needed because of the database model or the database management system. Thus, the latter schema is usually of lesser quality than the former.

Example 3-23.

We shall see in this section how the semantic schema of the university application can be converted into the relational schema of Figure ____.

(ii) Although only the latter schema (or its descendants after more conversions) will be used by the DBMS software, after conversion the former schema must be kept and maintained as a documentation of the application’s real world.

(iii) After conversion, the former schema is called the conceptual semantic schema of the latter physical schema or its descendants.

(iv) When the concepts of the real world change, the conceptual semantic schema must be changed first, and only then is the physical schema regenerated from the conceptual semantic binary schema by conversion.

This section presents a conversion algorithm of a semantic schema into a relational schema whose quality is among the highest possible for the Relational Model, provided the original semantic schema is of high quality.

This algorithm can be performed manually by the database designer. Alternatively, an automatic tool can be used to perform all the busywork, while prompting the database designer for intelligent decisions (and using defaults when the designer fails to provide such a guidance). One such tool has been developed at the Florida International University and the University of California by the author and his students.
3.4.2. Composition and split of relations

Two auxiliary definitions of terminology that will be used in the conversion algorithm follow.

**Composition of relations**

Let the range of Relation \( R_1 \) be the domain of Relation \( R_2 \).
Relation \( R \) is the composition of \( R_1 \) on \( R_2 \) if
\[
xRy \iff \text{exists } z \text{ such that } xR_1z \text{ and } zR_2y .
\]

**Example 3-24.**
Consider two relations:
- \( \text{the-course} \rightarrow \text{relation from COURSE-OFFERING to COURSE } \) (\( m:1 \))
- \( \text{name} \rightarrow \text{relation from COURSE to String } \) (\( 1:1 \))

The composition of \( \text{THE-COURSE} \) on \( \text{NAME} \) is:
- \( \text{the-name-of-the-course} \rightarrow \text{attribute of COURSE-OFFERING, range: String } \) (\( m:1 \))

**Relation-split** — conversion of a schema having a relation \( R \) into another schema having, instead of \( R \), a new abstract category \( C \) and two total functional relations \( R_1, R_2 \), whose domain is \( C \), such that:
There is a fact \( xRy \)
if and only if
there exists an object \( z \) in \( C \) for which \( zR_1x \) and \( zR_2y \).

Rishe-DDS p. 3.14
Figure 3-5. Relation $R$ is split into a category $C$ and two relations, $R_1$ and $R_2$. Every relationship $x - y$ is broken into $x - z$ and $z - y$.

Example 3-25.

If due to an implementational restriction we may not have a $m:m$ relation:

then we can split it into:

Rische-DDS p. 3.15
This split necessitates additional integrity constraints:

- Both new relations are total.
- For any combination of an instructor and a department there is at most one object in work.

The latter constraint is more rigorously formulated in calculus, as follows.

for every \( w \) in WORK:

for every \( v \) in WORK:

if \( w.\text{the-instructor} = v.\text{the-instructor} \) and  
\( w.\text{the-department} = v.\text{the-department} \)

then \( w = v \).

The composition of relations and relation-split can be regarded as userviews.*

* The following is a formal definition of the composition in Predicate Calculus for the inference rules of userviews. Here, the category \( C \) is the range of \( R_1 \).

userview relation: \( x R y \)

where exists \( z \) in \( C \):

\((x R_1 z \text{ and } z R_2 y)\)

The following is a formal definition of the relation-split in Predicate Calculus for the inference rules of userviews.

userview category \( C \) \( (R_1 : x, R_2 : y) \) where \( x R y \)
The following subsections present the conversion algorithm

### 3.4.3. Determination of keys

**Step 1. Choose a key for every abstract category**, excluding subcategories of other categories, as follows.

- **a. (single-attribute key)**
  
  *if* the category has an attribute which is 1:1, time-invariant, and total, *then* let that attribute be the key;

- **b. ("forced" single-attribute key)**
  
  *else if* the category has an attribute which can be implementationally restricted to be 1:1, time-invariant, and total, without very harmful alteration of the real world, *then* make that attribute into a key (declare the implementational restriction);

  **Example 3-26.**

  *name-key of COURSE*

  It is not a very far reaching alteration of the real world to make this implementation restriction: "Every course has exactly one name, and this name may never be changed."

- **c. (multiattribute key)**
  
  *else if* the category has a collection of attributes which are time-invariant and total, and jointly identify all the objects in the category, *then* let a minimal such collection be the key;

  **Example 3-27.**

  *(season-in-key, year-in-key) of QUARTER*

- **d. ("forced" multiattribute key)**
  
  *else if* the category has a collection of attributes which can be implementationally restricted to be time-invariant and total, and to jointly identify all the objects of the category, without very harmful alteration of the real world, *then* make a minimal such collection of attributes into a key;

- **e. (inferred key)**
  
  *else if* a collection of attributes can be inferred from the information existing in the schema and from keys of other categories, so that
  
  - these attributes can be implementationally restricted, without very harmful alteration of the real world,
    
    (i) to be time-invariant and total, and

Rishe-DDS p. 3.17
(ii) to jointly identify all the objects of the category, then

(i) choose a minimal such collection of inferable attributes;
(ii) add to the schema those attributes from the collection which are not already in the schema;
(iii) make this collection of attributes into a key (declare the implementational restrictions);
(iv) convert the inference rule of these attributes into constraints. (Since these will now be new attributes, their values will be updated by the users with possible inconsistency relative to the information from which these attributes are inferable.)

Example 3-28.
To obtain a key for \textit{DEPARTMENT} we alter the real world slightly: we require every department to have at least one name; we shall call the first name ever given to a department the \textit{main-name}, and we require that the \textit{main-name} of a department may never be changed. We add the new attribute

\begin{itemize}
    \item \textit{main-name-key} –attribute of \textit{DEPARTMENT}, range: \textit{String} (1:1)
\end{itemize}

and the constraint

\textbf{for every} \texttt{d in DEPARTMENT:}

\begin{itemize}
    \item \texttt{d NAME d.MAIN-NAME-key}
\end{itemize}

Note: In conjunction with the implicit constraint \texttt{-key}, the above constraint means that the main-name is the first name ever given to the department, and that it will remain the department’s name forever.

Example 3-29.
More characteristic examples of inferred keys are for the categories \textit{COURSE-OFFERING} and \textit{COURSE-ENROLLMENT}. These will be given and generalized after we have a key for \textit{PERSON}.

\underline{f. (enumerator ID key)}

\textit{else} create a new external enumeration for the objects in the category (thus altering the real world) and add it as an attribute, which will be the chosen key.

Example 3-30.
The key of \textit{PERSON} will be a new attribute \textit{id-key}.
Pragmatically, a program should be written to generate new values of an *enumerator id key*. These numbers will be assigned by the user to the new objects of the category. The numbers may not be reused when an object is removed. The numbers themselves should bear no correlation to the other information in the database, since the other information may change in time, while the key is time-invariant.

It is also advisable not to assign the numbers sequentially but rather in an arbitrary sequence. Otherwise, the irrelevant information on the ‘‘seniority’’ of objects will be hidden in the ID. Any hidden information will be abused by the application programmers. Since it is not always possible to update such hidden information correctly, the programs will not produce the expected results in some special cases.

*Note:* The step of finding keys is performed simultaneously for all the categories, since we might need to know the key of one category in order to find a key of a related category.

**Example 3-31.**

An *inferred key* of *COURSE-OFFERING* can be obtained when keys for *QUARTER, COURSE,* and *PERSON* have been chosen. The inferred key of *COURSE-OFFERING* will be

{the name of the course, the year of the quarter, the season of the quarter, ID of the instructor}

Hence, we add four new attributes to *COURSE-OFFERING*. The *inferred* key of *COURSE-ENROLLMENT* will be five new attributes

{ID of the student, the key of the offering}.

(The "key of the offering" consists of four attributes. Thus, there is a total of five attributes in the key of *COURSE-ENROLLMENT.*)

**Example 3-32.**

The category *COURSE-OFFERING* is now:

```
COURSE OFFERING

instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String
```

The above is an example of the prevalent case of an *inferred key*. The following is a generalization of this example.

Rishe-DDS p. 3.19
Assume that a category $C$ is the domain of total functional relations $f_1, \ldots, f_n$ which jointly identify all the objects of the category.

**Example 3-33.**

Every course offering is uniquely identified by its instructor, course, and quarter. Thus, the total functional relations

$$\text{THE-INSTRUCTOR, THE-COURSE, THE-QUARTER}$$

jointly identify all the objects of their domain, the category \text{COURSE-OFFERING}.

The above assumption means that there is an integrity constraint

$$\text{for every } x \text{ in } C:$$
$$\quad \text{for every } y \text{ in } C:$$
$$\quad \quad \text{if } x.f_1 = y.f_1 \text{ and } \cdots \text{ and } x.f_n = y.f_n$$
$$\quad \quad \text{then } x = y$$

**Example 3-34.**

$$\text{for every } x \text{ in COURSE-OFFERING:}$$
$$\quad \text{for every } y \text{ in COURSE-OFFERING:}$$
$$\quad \quad \text{if}$$
$$\quad \quad \quad x.\text{THE-INSTRUCTOR} = y.\text{THE-INSTRUCTOR} \text{ and}$$
$$\quad \quad \quad x.\text{THE-COURSE} = y.\text{THE-COURSE} \text{ and}$$
$$\quad \quad \quad x.\text{THE-QUARTER} = y.\text{THE-QUARTER}$$
$$\quad \quad \quad \text{then } x = y$$

In this case, once the keys of the ranges of the functional relations $f_1, \ldots, f_n$ are known, a key of $C$ can be inferred from them. Let the keys of the ranges be $k_1, \ldots, k_n$. Let $k_i$-of-$f_i$ be the set of inferred attributes obtained by the composition of the attributes comprising the key $k_i$ and the relation $f_i$.

**Example 3-35.**

There are three such sets of inferred attributes for the category \text{COURSE-OFFERING}:

- id-of-the-instructor
• the-name-of-the-course
• the-year-of-the-quarter, the-season-of-the-quarter

The key of $C$ is contained in the union of compositions of the relations $f_i$ onto the keys of their ranges, that is,

$$\{(k_1 \text{ of } f_1), \ldots, (k_n \text{ of } f_n)\}$$

Notice that the key of $C$ is contained in the above union of compositions. Usually the key of $C$ is equal to that union of compositions but sometimes it is properly contained.

Example 3-36.

Let us change the meaning of COURSE-OFFERING. Now, it does not have to occur in one particular quarter but can last several quarters, as long as the quarters are within one academic year (Figure 3-6). There are two relations between offerings and quarters:

- `beginning-quarter` → relation from COURSE-OFFERING to QUARTER (m:1,total)
- `ending-quarter` → relation from COURSE-OFFERING to QUARTER (m:1,total)

The key of COURSE-OFFERING is properly contained in

\[
\{\text{the name of the course; } \\
\text{the year and season of the beginning quarter; } \\
\text{the year and season of the ending quarter; } \\
\text{ID of the instructor}\}
\]

The attribute `THE-YEAR-OF-THE-ENDING-QUARTER` is not a part of the key, since this attribute is not needed for identification of the offerings. For a given beginning quarter and the season of the ending quarter, we can deduce the year of the ending quarter, since we know that the offering is within one academic year.
Figure 3-6. The case of multiquarter course offerings limited to one academic year. The key is shown.

Example 3-37.

The semantic schema of the university application has been converted so far into the schema on the following page.
Figure 3-7. The university schema with keys.
3.4.4. Disjointness of categories

Step 2. Convert the intersecting abstract categories into disjoint categories by the following procedure for every group of intersecting categories.

Example 3-38.
Here is a group of intersecting categories: STUDENT, INSTRUCTOR, PERSON.

A. Consider a complete group of categories so that every category outside the group is disjoint from every category in the group.

Let $C$ denote the union of all the categories in the group. If such a category $C$ does not already exist in the schema, then add it.

Let $S_1, S_2, \ldots, S_n$ be the other categories in the group. (All of them are direct or indirect subcategories of $C$.)

Example 3-39.
$C = \text{PERSON}, S_1 = \text{INSTRUCTOR}, S_2 = \text{STUDENT}.

Let

$$S_0 = C - \bigcup_{i=1}^{n} S_i$$

$S_0$ is the hypothetical category consisting of the objects of $C$ which do not belong to any of the subcategories. The category $S_0$ is considered in order to ensure that no information is lost during the conversion. It is not added to the schema at this time. It may or may not be added to the schema at a later step, depending on decisions made at that step.

Example 3-40.
If there may be other persons in addition to instructors and students, then

$S_0 = \text{OTHER-PERSON}$

Otherwise, $S_0 = \emptyset$, and it would not have to be added to the schema at any step.

In the continuation of this case study in the examples we will assume the latter case: no "other persons."

B. Estimate the intersection factors $\pi$ and $\rho$.

In order to choose the best way of conversion, we shall need to estimate the following quantities.
Example 3-41.

For the above group of intersecting categories, the choice of the method to eliminate the intersection of the categories will depend on the correlation of two parameters:

- The percentage of people who are both students and instructors, $\pi$
- The percentage of relations specific to students or instructors among all the relations which can be relevant to persons, $\rho$

$$\rho = \frac{\text{number of relations whose domain or range is } S_1 \text{ or } \ldots \text{ or } S_n}{\text{number of relations whose domain or range is } C \text{ or } S_1 \text{ or } \ldots \text{ or } S_n}$$

Example 3-42.

For the group of the previous examples, $\rho = \frac{5}{10}$.

$$\pi = \frac{\text{expected total number of objects in the intersections}}{\text{expected total number of objects in } C}$$

The above formula is rather informal.*

Example 3-43.

To estimate $\pi$, we have to predict the future of our database. It is reasonable to assume that about 5 percent of all persons would be simultaneously students and instructors, so $\pi = 0.05$.

* More formally:

$$\pi = \frac{\text{expected cardinality of } \bigcup_{i \neq j} (S_i \cap S_j)}{\text{expected cardinality of } C}$$

Rishe-DDS p. 3.25
Example 3-44.
If we had several intersecting categories, we would count all the intersections:

![Venn diagram showing instructors, students, alumni, and staff]

C. Select the best conversion into disjoint categories.

Example 3-45.
Consider the nondisjoint categories INSTRUCTOR and STUDENT, which are subcategories of the category PERSON=INSTRUCTOR ∪ STUDENT. The following are several possibilities of conversion. We will later select the best of the possibilities, depending on the circumstances.

a. Conversion into one category (Union)

Example 3-46.
Substitute the whole group of categories by their union, the category PERSON. This category will serve as the domain or the range for all the relations whose domain or range was one of the original categories. In addition, this category will have two Boolean attributes, IS-AN-INSTRUCTOR and IS-A-STUDENT, associating the value TRUE with objects representing instructors and students respectively.

PERSON

- id-key: Integer 1:1
- last-name: String
- first-name: String
- birth-year: 1870..1990
- address: String
- is-a-student: Boolean
- is-an-instructor: Boolean

- major —relation from PERSON to DEPARTMENT (m:1)

Risque-DDS p. 3.26
minor -> relation from PERSON to DEPARTMENT (m:1)

the-instructor -> relation from COURSE-OFFERING to PERSON (m:1)

the-student -> relation from COURSE-ENROLLMENT to PERSON (m:1)

works-in -> relation from PERSON to DEPARTMENT (m:m)

b. Conversion into artificially disjoint categories of Events

Example 3-47.
Substitute these categories by two disjoint categories of events: Event-of-being-a-STUDENT and Event-of-being-an-INSTRUCTOR (usually abbreviated just STUDENT and INSTRUCTOR, but the meaning of the full names is intended).

An instructor who is also a student will be represented by two distinct objects of the aforementioned categories.

The objects of the new categories are not persons but rather their “hats” — a person may have two “hats”: one as an instructor and one as a student. The two categories of “hats” are disjoint.

The relations whose domain or range is the category PERSON, for example, the relation ADDRESS, will be replaced by two relations having the new categories as their domains or ranges, such as the relations STUDENT’S-ADDRESS and INSTRUCTOR’S-ADDRESS.

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>INSTRUCTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>id-key: Integer 1:1</td>
<td>id-key: Integer 1:1</td>
</tr>
<tr>
<td>last-name: String</td>
<td>last-name: String</td>
</tr>
<tr>
<td>first-name: String</td>
<td>first-name: String</td>
</tr>
<tr>
<td>address: String</td>
<td>address: String</td>
</tr>
</tbody>
</table>

It may appear that by introducing the categories of “hats” we have succeeded in fooling the system. Actually, we have fooled ourselves. Without understanding the relationship between two hats of one person, the system will not be able to correctly interpret some queries of naive users and may cause inconsistency in the stored information and other problems:

- When the address of a person is updated, it may get updated in one category but not in the other. The database will become inconsistent.

Rishe-DDS p. 3.27
A naive query like "How many people are there?" will involve double count of persons who are instructors and students simultaneously.

c. Conversion into Union + Events

Example 3-48.

As Figure 3-8 shows, we can retain the category PERSON with all its relationships and define two categories of events which will inherit all the relationships of STUDENT and INSTRUCTOR, and additionally will have keys and special 1:1 relationships with the category PERSON:

- PERSON —category (retains its relations from the binary schema)
- Event-of-being-a-STUDENT —category (inherits all the relations of STUDENT)
- Event-of-being-an-INSTRUCTOR —category (inherits all the relations of INSTRUCTOR)
- student-person —relation from Event-of-being-a-STUDENT to PERSON (1:1,total)
- instructor-person —relation from Event-of-being-an-INSTRUCTOR to PERSON (1:1,total)

Figure 3-8. Union + events
Example 3-49.

To further explore the differences between the three approaches, consider the formulation of the query “Print the names of all the students.”

**Events**:

```
get s.LAST-NAME
where s is a STUDENT
```

**Union**:

```
get s.LAST-NAME
where
  s is a PERSON and
  s.IS-A-STUDENT
```

**Union + Events**:

```
get p.LAST-NAME
where
  p is a PERSON and
  exists s in STUDENT:
    s.ID-key = p.ID-key
```

**Relative disadvantages of each approach**

The principal disadvantage of **Events** is the **redundancy**. For example, the birth-year of an instructor who is also a student has to be logically represented twice in the database, which can cause inconsistency and other problems.

The principal disadvantages of **Union** are the **unnaturalness** of the schema and the **under-coverage of integrity constraints**. For example, an additional integrity constraint has to be defined to prevent association of a nonstudent instructor with a major department of studies. Another important deficiency is the **null-values**, causing significant problems in formulation of queries. (We say that “p.MAJOR is null” if the person p is not related to any department by the relation MAJOR.)

The principal disadvantages of **Union + Events** are the **unnaturalness** of the schema and significant difficulties in the formulation of queries and other operations. These difficulties, however, can be overcome by the use of userviews which would conveniently redefine the concepts of the schema. This requires that the DBMS provide a high-level support for userviews, including the capability to specify updates through userviews. Most relational DBMS, however, do not provide sufficient support of userviews.

**Conclusion**
Unless the DBMS provides sufficient support for userviews as discussed above, we have to exclude the **Union + Events** approach.

Both other approaches, **Union** and **Events**, would result in low-quality schemas, but the relational database designer has to choose the better of the two.

**Example 3-50.**

The choice should usually depend on the correlation of two parameters: the percentage of people who are both students and instructors, \( \pi \), and the percentage of relations specific to students or instructors among all the relations which can be relevant to persons, \( \rho \).

The relative redundancy in **Events** increases when \( \pi \) increases and when \( \rho \) decreases. The unnaturalness and the undercoverage of constraints in **Union** increase when \( \pi \) decreases and when \( \rho \) increases.

The following provides a decision criteria for an arbitrary group of categories. The decision is made according to the \( \pi:\rho \) ratio. A comparison quotient of 0.6 is suggested, which is quite often reasonable, as has been shown by analysis of a class of databases. If \( \pi/\rho > 0.6 \) then the **Union** approach would usually be preferable. In some special cases, however, the database designer should consider a different comparison quotient. The number 0.6 is only a ‘rule of thumb.’

When there is chain of sub-sub-categories, the approach **Events** becomes too complicated and is not recommended. It is, however, the most natural approach in the majority of situations, because in the majority of cases \( \pi \) is small, the subcategory hierarchy is rather flat (no sub-sub-categories), and the DBMS does not provide a sufficient support for userviews.

D. Convert the group of categories into disjoint categories.

a. **if** the DBMS provides a high level support for userviews, including specification of updates, **then** (**Union + Events**):

(i) Substitute every direct or indirect subcategory \( S \) of \( C \) in the schema being converted by the category **Event-of-being-a-[n]-S**. Each object in this new category is an event of membership in the category \( S \); that is, if \( x \) is an \( S \) then "\( x \) is an \( S \)" is one element in **Event-of-being-an-S**. (The categories of events are disjoint. For simplicity, the former names \( S \) may be kept but the new meaning is assumed.)

**Example 3-51.**

- [Event-of-being-a-]STUDENT – category
- [Event-of-being-an-]INSTRUCTOR – category

Risque-DDS p. 3.30
Example 3-52.
If we also had

- TENURED-FACULTY — subcategory of INSTRUCTOR

then we would convert it into

- Event-of-being-TENURED-FACULTY — category

(ii) Retain the category $C$.

Example 3-53.

- PERSON — category

(iii) Connect every new category of events $S$ to each immediate supercategory of $S$ by a new relation. Specify integrity constraints that these new relations are one-to-one and total.

Example 3-54.

- student-person — relation from Event-of-being-a-STUDENT to PERSON (1:1, total)

(iv) Let every new category of events $S$ have all the relations that the former category $S$ had.

Example 3-55.

- major — relation from Event-of-being-a-STUDENT to DEPARTMENT (m:1)

(v) Specify and add a key for every category of events $S$. The simplest way to do this is to inherit the key of $C$.

Example 3-56.

- id-key — attribute of Event-of-being-a-STUDENT, range: Integer (1:1)

b. **else if** $\pi/\rho > 0.6$ or there is a chain of sub-sub-categories **then** (Union):

(i) Replace the whole group of categories by one category $C$. 

Rishe-DDS p. 3.31
Example 3-57.

□ PERSON –category

(ii) Bring all the relations exiting or entering the former subcategories to $C$.

Example 3-58.

□ the-student → relation from COURSE-ENROLLMENT to PERSON (m:1)

(iii) Add to $C$ total Boolean attributes named is-a[n]→S for every direct and indirect subcategory $S$ of $C$ in the schema being converted.

Example 3-59.

□ is-a-student → attribute of PERSON, range: Boolean (m:1,total)
□ is-an-instructor → attribute of PERSON, range: Boolean (m:1,total)

(iv) Add an integrity constraint stating that any object of $C$ may participate in a former $S$’s corresponding relation only if the respective function is-an-S gives TRUE.

Example 3-60.

for every p in PERSON:
    if exists d in DEPARTMENT:  p WORKS-IN d
    then p.IS-AN-INSTRUCTOR

Example 3-61.

for every p in PERSON:
    if not (p MAJOR null)
    then p.IS-A-STUDENT

(v) Whenever there are attributes is-an-S$_1$ and is-an-S$_2$, where $S_1$ is a subcategory of $S_2$ in the original schema, add a constraint enforcing that in terms of the new attributes.
Example 3-62.
If we had:

- **UNDERGRADUATE** — subcategory of **STUDENT**

then we would add a constraint:

for every s in PERSON:

if s.IS-AN-UNDERGRADUATE

then s.IS-A-STUDENT

(vi) Whenever there are attributes is-an-\(S_1\) and is-an-\(S_2\), where \(S_1\) and \(S_2\) are disjoint in the original schema, add a constraint enforcing that in terms of the new attributes.

Example 3-63.
If the category **UNDERGRADUATE** was disjoint from the category **INSTRUCTOR**, then we would add a constraint:

for every s in PERSON:

if s.IS-AN-UNDERGRADUATE

then not (s.IS-AN-INSTRUCTOR)

c. else (Events):

(i) Substitute the categories \(S_1, \ldots, S_n\) by the corresponding \(n\) categories **Event-of-being-a-\(S_1\), \ldots, Event-of-being-a-\(S_n\)** of the events of membership in categories; that is, if \(x\) is an \(S_i\) then "\(x\) is an \(S_i\)" is one element in the category **Event-of-being-an-\(S_i\)**. (The categories \(S_i\) are disjoint. For simplicity, former names \(S_i\) may be kept but the new meaning is assumed.)

Example 3-64.
- **Event-of-being-a-STUDENT** — category

disjoint from

- **Event-of-being-an-Instructor** — category

(ii) If there are, or may be in the future, objects in \(C\) that do not belong to any of the subcategories \(S_1, \ldots, S_n\), then add a new category \(S_0\) to the schema. This will be the category of the objects that do not belong to any of the subcategories. This category is usually called **other-C**.
Example 3-65.

OTHER-PERSON — category

(iii) Replace every relation $R$ whose domain or range is $C$, by new relations of the same name as $R$ but having the categories $S_i$ as their domains or ranges. (The relation $R$ is partitioned into several relations according to the restricted domains or ranges $S_i$.)

Example 3-66.

birth-year — attribute of Event-of-being-a-STUDENT, range: 1870..1990 (m:1)

(iv) Eliminate the category $C$.

(v) Specify integrity constraints to prevent inconsistency of the redundant information:

```
“If an object $x$ of the category Event-of-being-an-$S_i$ has the same key values as an object $y$ of the category Event-of-being-an-$S_j$, then the other relations of $C$ (inherited by the categories of events) must be equal for $x$ and $y.”
```

Example 3-67.

We choose this alternative (Events) for the intersecting group of the subcategories of PERSON in the case-study database.

The schema now has redundancy, which should be controlled by an integrity constraint, if possible. The integrity constraint is

for every $s$ in Event-of-being-a-STUDENT:

for every $i$ in Event-of-being-an-INSTRUCTOR:

if

(s.ADDRESS $\neq$ i.ADDRESS or
s.LAST-NAME $\neq$ i.LAST-NAME or
s.FIRST-NAME $\neq$ i.FIRST-NAME or
s.BIRTH-YEAR $\neq$ i.BIRTH-YEAR)

then s.ID-key $\neq$ i.ID-key

(Note: The constraint could have been written without negations, “in a positive spirit,” but then the meaning of the absent values could be misinterpreted.)
Example 3-68.
The semantic schema of the university application has been converted so far (by Events) into the schema on the following page.

3.4.5. Removal of relations

Step 3. Convert every proper 1:m or m:m relation whose range is a concrete category into a new abstract category with its two functional relations through a relation-split.

Example 3-69.
Instead of the relation

- name — relation from DEPARTMENT to String (1:m)

we shall have

- DEPARTMENT-NAMING — category
- the-department — relation from DEPARTMENT-NAMING to DEPARTMENT (m:1,total)
- the-name — relation from DEPARTMENT-NAMING to String (1:1,total)

Step 4. Convert every 1:m relation into an m:1 relation by changing its direction and its name.

Example 3-70.
We do not have such relations in the university schema. If we assume we have the relation

- provides — relation from DEPARTMENT to COURSE (1:m)

then we would change it into

- the-department-providing-the-course — relation from COURSE to DEPARTMENT (m:1)

Step 5. Convert every proper many-to-many relation into a category and two functional relations through a relation-split.

Rishe-DDS p. 3.35
Event of being a STUDENT

last-name: String
first-name: String
birth-year: 1870..1990
address: String
id-key: Integer 1:1

Event of being an INSTRUCTOR

last-name: String
first-name: String
birth-year: 1870..1990
address: String
id-key: Integer 1:1

DEPARTMENT

name: String 1:m
main-name-key: String 1:1

COURSE OFFERING

instructor-id-in-key: Integer
course-name-in-key: String
department-name-in-key: String
season-in-key: String
year-in-key: 1980..1995

COURSE

name-key: String 1:1

COURSE ENROLLMENT

final-grade: 0..100
student-id-in-key: Integer
course-name-in-key: String
season-in-key: String
year-in-key: 1980..1995

QUARTER

season-in-key: String
year-in-key: 1980..1995

major (m:1)

minor (m:1)

works in (m:m)

works in (m:1)

works in (m:1)

Figure 3-9. The university schema with the categories made artificially disjoint.

Example 3-71.

We split the relation WORKS-IN into a new category WORK and its two functional relations THE-DEPARTMENT and THE-INSTRUCTOR.
Example 3-72.
If we had the following m:m relation

```
INSTRUCTOR                     COURSE
                        is able to teach
(m:m)
```

then we would split it into

```
INSTRUCTOR
```
```
ABILITY TO TEACH
```
```
the instructor
(m:1)
```
```
the course
(m:1)
```
```
COURSE
```
Figure 3-10. The university schema after all the relation splits have been performed.
Step 6. Choose a key for every category produced through a relation-split as follows.

For every category which was obtained through a relation-split, a key is contained in the union of the compositions of its two functional relations on the keys of their ranges.

Example 3-74.
The key of WORK is two new attributes of this category:
{main-name of the department, instructor-id of the instructor}

Example 3-75.
The key of DEPARTMENT-NAMING is contained in
{the-name, main-name of the department}
Since the-name is 1:1, the key of DEPARTMENT-NAMING is {the-name}.

Example 3-76.
The semantic binary schema of the university application has been converted so far into the schema in Figure 3-11.
Figure 3-11. The university schema after the relation splits have been performed and keys have been chosen for every category.
**Step 7.** Replace every m:1 relation \( f \) whose range is an abstract category by the composition of \( f \) on the chosen key of its range, that is, by attributes \( b_1, \ldots, b_n \), where \( x.b_i = (x.f).a_i \), and \( a_1, \ldots, a_n \) is the chosen key of \( f \)’s range.

*Example 3-77.*

Instead of

- **major** → relation from \( STUDENT \) to \( DEPARTMENT \) \((m:1)\)

we shall have

- **major-dept-main-name** → attribute of \( STUDENT \), range: \( String \) \((m:1)\)

**Step 8. Remove redundant nonkey attributes.**

From every category **remove attributes** which are not in the key but are inferable from other attributes of the same category.

These attributes would usually have resulted from a “blind” application of this algorithm, particularly

a. A nonkey attribute which is always equal to an attribute in the key.

*Note:* It is possible that step (7) brought to a category \( C \) an attribute \( b \) which is always equal to an attribute in the key of \( C \).

b. A redundant Boolean attribute brought in step (1):

Suppose:

- We were converting intersecting categories \( C, S_1, \ldots, S_n \) into disjoint categories.
- We replaced them by one **Union** category \( C \).
- one of the categories \( S_i \) was disjoint from all of the rest \( S_j \)’s.

Then the new attribute is-an-\( S_i \) is inferable from the rest of the attributes is-an-\( S_j \).

This attribute should be removed from the schema. (It may be present in a userview, where it would be an inferred attribute.)

*Example 3-78.*

If we had

- **ILLITERATE** → subcategory of \( PERSON \) (disjoint from \( INSTRUCTOR \) and from \( STUDENT \) )

and furthermore, there were no other persons but students, instructors, and illiterate persons, then the attribute **is-illiterate** would be derivable:

for every \( p \) in \( PERSON \):
p.IS-ILLITERATE =
(not p.IS-A-STUDENT and not p. IS-AN-INSTRUCTOR)

Note: The removal of several attributes should not be performed simultaneously. Otherwise, two attributes mutually inferable, but not inferable from the rest, might be removed.

3.4.6. Integrity constraints

Step 9. Translate the integrity constraints into the terms of the new schema:

a. The constraints of the original schema.

b. The additional constraints accumulated during the conversion process.

Example 3-79.
The semantic schema of the university application has been converted into the relational schema of Figure ___.

Reference Schemas

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.
Figure Ref-1. A semantic schema for a university application.
**Figure Ref-2.** A relational schema for the university application.
4. RELATIONAL DATABASE LANGUAGES

This chapter describes relational database languages. Sections 1 and 2 show examples of how the languages of Chapter 2 (the fourth-generation and the logic-based languages) apply to the relational databases. The principal example of Section 1 discusses the principles of writing a transaction-processing program for an application. The optional Section 3 defines the Relational Algebra. Section 4 describes SQL, a popular commercial language related to the Relational Predicate Calculus of Section 2. The expressive power and equivalence of relational languages is discussed in Section 5.

4.1. Fourth-generation Programming

The structured extension of Pascal for relational databases is the same as the extension of Pascal for semantic databases but is used for the relational schema. (Every relational schema is a binary schema. Thus, every language used for the Binary Model can be used for the Relational Model.)

Example 4-1.

The university has decided to expel all the students whose average grade is below 60 (out of 100). To prevent this wrong-doing to computer science students, the department offered a fictitious course, Computer-Pass, by Prof. Good, in which all computer science students are to receive a sufficient grade so as to not to be expelled, if possible.

The following program fabricates Prof. Good and the Computer-Pass course, enrolls students in this course, grades them accordingly, and prints the names of those computer science students whom this measure cannot help.

program Pass (Input, Output, UNIVERSITY-DB, UNIVERSITY-MASTER-VIEW);
var Computer-Pass-Course, Prof-Good, Good-Offer, computer-science-name, comp-science, cs-student, her-enrollment, fictitious-enrollment: ABSTRACT;
    the-grade, desired-grade, number-of-grades, total-of-grades, current-year: INTEGER;
begin
(* Get the current year from the standard input file. *)
    read (current-year);
transaction begin

(* Fabricate the course. *)

create new Computer-Pass-Course in COURSE;
Computer-Pass-Course.NAME-key := ‘Computer Pass’;

(* Fabricate the Prof. Good. *)

create new Prof-Good in INSTRUCTOR;
Prof-Good.LAST-NAME := ‘Good’;
Prof-Good.ID-key := 1234; (* Let’s hope that this fabricated ID number does not already belong to a legitimate instructor. Otherwise, an error will result. *)

(* Fabricate the offering. *)

create new Good-Offer in COURSE-OFFERING;
Good-Offer.COURSE-NAME-in-key := ‘Computer Pass’;
Good-Offer.INSTRUCTOR-ID-in-key := 1234;
Good-Offer.YEAR := current-year;
Good-Offer.SEASON := ‘Winter’
end;

(* The following two nested loops will be performed only once. Inside the body of the second loop, the variable comp-science will refer to the Computer Science Department. *)

for computer-science-name in DEPARTMENT-NAMING
where (computer-science-name.NAME-key = ‘COMPUTER SCIENCE’) do
for comp-science in DEPARTMENT
where (comp-science.MAIN-NAME-key = computer-science-name.MAIN-NAME)
do begin

(* Make believe that Prof. Good works in Computer Science. *)

(* In terms of the binary schema, relate: Prof-Good WORKS-IN comp-science *)

transaction begin
create new work in WORK;
work.INSTRUCTOR-ID-in-key := 1234;
work.DEPARTMENT-MAIN-NAME-in-key := comp-science.MAIN-NAME-key
end;

Rishe-DDS p. 4.2
for cs-student in STUDENT
        where (cs-student.MAJOR-DEPARTMENT-MAIN-NAME
              = comp-science.MAIN-NAME-key) do
begin (* the current computer science student *)
(* calculate this student’s current statistics: number-of-grades and total-of-
   grades *)
    number-of-grades := 0;
    total-of-grades := 0;
for her-enrollment in COURSE-ENROLLMENT where (her-enrollment.-
    STUDENT-ID-in-key = cs-student.ID-key and not her-enrollment
    FINAL-GRADE null) do begin
    the-grade := her-enrollment.FINAL-GRADE;
    number-of-grades := number-of-grades + 1;
    total-of-grades := total-of-grades + the-grade
end;
(* calculate the minimal desired grade in computer-pass course, solving the
   equation
   (total+x)/(number+1)=60 *)
    desired-grade := 60 * (number-of-grades + 1) — total-of-grades;
if desired-grade > 100 then
    (* the student cannot be helped. Print a message *)
    writeln (` The student ', cs-student.LAST-NAME, ` cannot be
              helped. Sorry!')
else if desired-grade > 60 then
    transaction begin
create new fictitious-enrollment in COURSE-ENROLLMENT;
fictitious-enrollment.STUDENT-ID-in-key := cs-student.ID-key;
fictitious-enrollment.COURSE-NAME-in-key := ‘Computer Pass’;
fictitious-enrollment.INSTRUCTOR-ID-in-key := 1234;
fictitious-enrollment.YEAR := 1990;
fictitious-enrollment.SEASON := ‘Winter’;
fictitious-enrollment.FINAL-GRADE := desired-grade
end (* transaction *)
end (* current student *)

Risque-DDS p. 4.3
4.1.1. Transaction processing

Many application programs process a continuous sequence of end-user requests for transactions. In each cycle, such a program accepts a request from an end-user, translates it from the data-entry form into the terms of the database, determines the validity of the request, and performs it.

Most relational DBMS are unable to automatically enforce the integrity constraints, particularly referential integrity. In this case, the busy-work of manual integrity validation must be performed by the application programs processing the user transactions.

Example 4-2.

The following program reads from the standard input a series of requests for enrolling students in classes. For every request, the program checks its integrity against the database. If the request is integral, the program performs the update. Otherwise, an error message is printed.

It is assumed that the DBMS is unable to automatically enforce the referential integrity.

program Enroll (Input, Output, University-database, University-master-userview);

var

student-id, instructor-id, year: Integer;
course-name, season: String;

student, instructor, course, quarter, offering, enrollment: ABSTRACT;
student-ok, instructor-ok, course-ok, quarter-ok, offering-ok, enrollment-ok: Boolean;

procedure Erroneous-transaction (explanation: String);

(* This procedure is called
   • from Transaction-error-handler, when an error is detected by the DBMS
   • from the program, when an error is detected by the program. *)

begin

writeln (`The enrollment request listed in the following line could not be granted for the reason: ', explanation);
writeln (student-id, instructor-id, course-name, year, season)

end

procedure Transaction-error-handler (error-description: String):

Rishe-DDS p. 4.4
begin
Erroneous-transaction (concatenate(‘System error: ’, error-description);
end
begin
while not eof(Input) do
transaction begin
(* Get a request for the next enrollment transaction *)
readln (student-id, instructor-id, course-name, year, season)
(* Check student ID *)
student-ok := false;
for student in STUDENT
    where (student.ID-key = student-id)
    do student-ok := true;
if not student-ok then
    Erroneous-transaction (‘No student known by the student-id submitted in the first field of the enrollment request’);
(* Check instructor ID *)
instructor-ok := false;
for instructor in INSTRUCTOR
    where (instructor.ID-key = instructor-id)
    do instructor-ok := true;
if not instructor-ok then
    Erroneous-transaction (‘No instructor known by the instructor-id submitted in the second field of the enrollment request’);
(* Check course name *)
course-ok := false;
for course in COURSE
    where (course.NAME-key = course-name)
    do course-ok := true;
if not course-ok then
    Erroneous-transaction (‘No course known by the course-name submitted in the third field of the enrollment request’);
(* Check quarter *)
quarter-ok := false;

for quarter in QUARTER
  where (quarter.YEAR-in-key = year and quarter.SEASON-in-key = season)
  do quarter-ok := true;
if not quarter-ok then
  Erroneous-transaction (‘No quarter known by the year and season submitted
  in the fourth and fifth fields of the enrollment request’);

(* Check the offering *)
offering-ok := false;

for offering in COURSE-OFFERING
  where (offering.INSTRUCTOR-ID-in-key = instructor-id and offering.
    COURSE-NAME-in-key = course-name and offering.YEAR-in-key = year
    and offering.SEASON-in-key = season)
  do offering-ok := true;
if instructor-ok and course-ok and quarter-ok and not offering-ok then
  Erroneous-transaction (‘No offering known by the instructor-id, course, year
  and season submitted in the second through fifth fields of the enrollment
  request’);

(* The following check of nonduplicate enrollment is not strictly necessary, since it can
be performed automatically by the DBMS, which knows to enforce the uniqueness
of the keys. Thus, the only practical reason for this test is to produce a better
message than what would be produced by the system by default. *)

(* Check that student is not already enrolled in the offering *)
enrollment-ok := true;

for enrollment in COURSE-ENROLLMENT
  where (enrollment.STUDENT-ID-in-key = student-id and enrollment.
    INSTRUCTOR-ID-in-key = instructor-id and enrollment.COURSE-
    NAME-in-key = course-name and enrollment.YEAR-in-key = year and
    enrollment.SEASON-in-key = season)
  do enrollment-ok := false;
if not enrollment-ok then
  Erroneous-transaction (‘The requested enrollment of the student already
  exists’)
if student-ok and instructor-ok and course-ok and quarter-ok and offering-ok and
enrollment-ok
then

Rishe-DDS p. 4.6
(* Insert the new enrollment *)

\begin{verbatim}
begin
create new enrollment in COURSE-ENROLLMENT;
enrollment.STUDENT-ID-in-key := student-id;
enrollment.INSTRUCTOR-ID-in-key := instructor-id;
enrollment.COURSE-NAME-in-key := course-name;
enrollment.YEAR-in-key := year;
enrollment.SEASON-in-key := season
end
\end{verbatim}

dead (* transaction *)
dead.

4.2. Logic for Relational Databases

Relational Calculus — Predicate Calculus, when used for the relational schema. (Since every relational schema is a binary schema, we already know Relational Calculus.)

---

**Example 4-3.**
What are the last names of all the students?
\begin{verbatim}
get s.LAST-NAME
where s is a STUDENT
\end{verbatim}

---

**Example 4-4.**
What are the distinct last names of the students? (No name may be printed twice.)
\begin{verbatim}
get n
where
exists s in STUDENT:
n = s.LAST-NAME
\end{verbatim}

---

Tuple-oriented Relational Calculus — Relational calculus with the following restriction: the quantification is done only on abstract categories (i.e., tables).

Among the languages of the Relational Model, more languages are based on the tuple-oriented Predicate Calculus than on the more general form of Predicate Calculus.

Rishe-DDS p. 4.7
Example 4-5.
The previous example was not in the tuple-oriented form because the variable \( n \) was implicitly quantified over the concrete category \( \text{String} \).
The following examples are in tuple-oriented form.

Example 4-6.
Has every student enrolled in at least one course in 1990?

\[
\text{for every } st \text{ in } \text{STUDENT}:
\quad \exists \text{ enrl in } \text{COURSE-ENROLLMENT}:
\quad ((\text{enrl.STUDENT-ID-in-key} = \text{st.ID-key}) \ \text{and} \ \text{enrl.YEAR}=1990))
\]

Example 4-7.
Who took Prof. Smith’s courses?

\[
\text{get student.LAST-NAME where}
\quad (\text{student is a STUDENT and}
\quad \exists \text{ enrl in } \text{COURSE-ENROLLMENT}:
\quad ((\text{enrl.STUDENT-ID-in-key} = \text{student.ID-key}) \ \text{and}
\quad \exists \text{ inst in } \text{INSTRUCTOR}:
\quad ((\text{inst.LAST-NAME} = \text{‘Smith’ and}
\quad \text{enrl.INSTRUCTOR-ID-in-key} = \text{inst.ID-key})))
\]

Example 4-8.
Print the average grade of every computer science student.

\[
\text{get student.LAST-NAME,}
\quad (\text{average enrollment.FINAL-GRADE}
\quad \text{where}
\quad \text{enrollment.STUDENT-ID-in-key} = \text{student.ID-key})
\quad \text{where}
\quad \text{student is a STUDENT and}
\]

Rishe-DDS p. 4.8
Example 4-9.
How many students are there in the university?
\[
\text{get (count std where std is a STUDENT)}
\]

Example 4-10.
What students have their average grade below 60?
\[
\text{get std.LAST-NAME}
\]
\[
\text{where std is a STUDENT and}
\]
\[
60 > \text{average enrl.FINAL-GRADE}
\]
\[
\text{where}
\]
\[
\text{enrl is a COURSE-ENROLLMENT and}
\]
\[
\text{enrl.STUDENT-ID-in-key = std.ID-key}
\]

4.3. *Relational Algebra

Relational Algebra is an algebraic language in which new tables are defined by applying operators to other tables.

This is a language of expressions. In it, a new table is defined as an expression involving original tables and operators.

The most important operators are:

- **Projection operator** creates a new table containing some of the columns of another table.
- **Join operator** combines the rows of the first table with "related" rows of the second table.
- **Selection operator** extracts some rows from a table according to a given condition on the values of the row.

Example 4-11.
\[
\{\text{The last names of the students born in 1975}\} = \]
\[
(project\text{-the-column-LAST-NAME}
\]
\[
(select\text{-the-rows-where-the-BIRTH-YEAR-is-1975}
\]
In this section, the operators of Relational Algebra are defined by inference laws in Predicate Calculus.

Consider two tables:

- table $T$, whose attributes are $A_1, \ldots, A_n$
- table $T'$, whose attributes are $A'_1, \ldots, A'_n$

1. The **projection operator** (. . .[attributes]) creates a new table containing some of the columns of another table.

   Let $F_1, \ldots, F_k$ be some of the attributes (columns) of table $T$. Then

   \[
   T [F_1, \ldots, F_k] = \text{get } F_1: v_1, \ldots, F_k: v_k \\
   \text{where exists } x \text{ in } T: \\
   x.F_1=v_1 \text{ and } \cdots \text{ and } x.F_k=v_k
   \]

   **Note:** When several tuples of $T$ have the same values in the columns being projected, only one row will appear in the result. Thus, the resulting table may have fewer tuples than $T$.

**Example 4-12.**

A list of the distinct last names of the students =

\[
STUDENT[LAST-NAME] = \text{get } LAST-NAME: name \\
\text{where } \\
\text{exists } x \text{ in } STUDENT: \\
x.LAST-NAME = name
\]

**Example 4-13.**

Last names and majors of all the students =

\[
STUDENT[LAST-NAME, MAJOR-DEPARTMENT-MAIN-NAME] = \text{get } LAST-NAME: name, MAJOR-DEPARTMENT-MAIN-NAME : major \\
\text{where } \\
\text{exists } x \text{ in } STUDENT:
\]

Rishe-DDS p. 4.10
Example 4-14.

We can define an inferred table \( \text{STUDENT-BASIC} \) containing all the information from the table \( \text{STUDENT} \) except the departments:

\[
\text{STUDENT-BASIC} = \text{STUDENT} [\text{ID-key, LAST-NAME, FIRST-NAME, BIRTH-YEAR, ADDRESS}]
\]

2. The **renaming operator** \([\ldots[\text{attribute/new-name}]\] changes the name of a column in a table.

\[
T[A_i / A_j] =
(* \text{Copy the attributes } A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n; \text{rename the attribute } A_i. *)
\]

\[
\text{get } A_1: x.A_1, \ldots, A_{i-1}: x.A_{i-1}, \overline{A_i}: x.A_i, A_{i+1}: x.A_{i+1}, \ldots, A_n: x.A_n
\]

where \( x \) is a \( T \)

Example 4-15.

A table just like \( \text{STUDENT} \), with ‘FAMILY-NAME’ column title instead of ‘LAST-NAME’:

\[
\text{STUDENT} [\text{LAST-NAME/FAMILY-NAME}]
\]

3. The **cartesian product operator** \( \times \)

For every row of the first operand and for every row of the second operand, the product operator produces the concatenation of the two rows.

The number of rows in the result =

\[
(\text{the number of rows in the first operand} \times \text{the number of rows in the second operand})
\]

The number of columns in the result =

\[
(\text{the number of columns in the first operand} + \text{the number of columns in the second operand})
\]

This operation is syntactically erroneous when the two tables have a common attribute.

Cartesian product =

\[
T \times T' =
\]

Rishe-DDS p. 4.11
4. **Set operators**

The following operators are defined only when the two tables have the same attributes.

a. **Union of tables** produces all the rows of the first table and all the rows of the second table.

\[
T \cup T' = \text{get } A_1:v_1, \ldots, A_n:v_n \text{ where exists } x \text{ in } T:
\]
\[
x.A_1 = v_1 \text{ and } \cdots \text{ and } x.A_n = v_n
\]

or exists \(x \text{ in } T'\):
\[
x.A_1 = v_1 \text{ and } \cdots \text{ and } x.A_n = v_n
\]

Example 4-16.
All the persons =

\[
\text{STUDENT-BASIC} \cup \text{INSTRUCTOR} = \text{get } \text{ID-key} : \text{id}, \ldots, \text{ADDRESS} : \text{addr} \text{ where exists } x \text{ in STUDENT-BASIC:}
\]
\[
x.\text{ID-key} = \text{id} \text{ and } \cdots \text{ and } x.\text{ADDRESS} = \text{addr}
\]

or exists \(x \text{ in INSTRUCTOR:}
\]
\[
x.\text{ID-key} = \text{id} \text{ and } \cdots \text{ and } x.\text{ADDRESS} = \text{addr}
\]

b. **Intersection of tables** produces the rows which appear in both tables.

\[
T \cap T' = \text{get } A_1:v_1, \ldots, A_n:v_n \text{ where exists } x \text{ in } T:
\]
\[
x.A_1 = v_1 \text{ and } \cdots \text{ and } x.A_n = v_n
\]

and exists \(x \text{ in } T'\):
Example 4-17.
Instructors who are students = 
\[ \text{INSTRUCTOR} \cap \text{STUDENT-BASIC} \]

\[ x.A_1 = v_1 \quad \text{and} \quad \cdots \quad \text{and} \quad x.A_n = v_n \]

c. The **difference of tables operator** (−) produces the rows of the first table which do not appear in the second table.

\[ T − T' = \]

\[ \text{get } A_1:v_1, \ldots, A_n:v_n \]

where

\[ \text{exists } x \text{ in } T: \]

\[ x.A_1 = v_1 \quad \text{and} \quad \cdots \quad \text{and} \quad x.A_n = v_n \]

\[ \text{and not exists } x \text{ in } T': \]

\[ x.A_1 = v_1 \quad \text{and} \quad \cdots \quad \text{and} \quad x.A_n = v_n \]

Example 4-18.
Instructors who are not students = 
\[ \text{INSTRUCTOR − STUDENT-BASIC} \]

5. The **selection operator** ([condition]) extracts some rows from a table according to a given condition for the values of the row.

Let \( F_1, \ldots, F_k \) be some of the attributes of \( T \).

Let \( \text{boolexp} (v_1, \ldots, v_k) \) be a Boolean expression with \( k \) variables.

Then

\[ T [\text{boolexp}(F_1, \ldots, F_k)] = \]

\[ \text{get } x.A_1, \ldots, x.A_n \]

where

\[ x \text{ is a } T \text{ and} \]

\[ \text{boolexp} (x.F_1, \ldots, x.F_k) \]

Risque-DDS p. 4.13
Example 4-19.
The student whose first name is Mary =
\[ \text{STUDENT} \{ \text{FIRST-NAME}=\text{‘Mary’} \} \]

Example 4-20.
The instructor whose name is Chung and who is not a student (as distinguished from another Chung who is both an instructor and a student.)
All the instructors whose name is ‘Chung’ =
\[ \text{INSTRUCTOR} \{ \text{LAST-NAME}=\text{‘Chung’} \} \]
The nonstudent instructor(s) whose name is ‘Chung’ =
\[ \text{INSTRUCTOR} \{ \text{LAST-NAME}=\text{‘Chung’} \} - \text{STUDENT-BASIC} \]

Example 4-21.
Names of the instructors teaching databases.
All the combinations of instructors and offerings (including the unrelated ones) =
\[ (\text{INSTRUCTOR} \times \text{COURSE-OFFERING}) \]
(*product*)
All combinations of instructors and their offerings =
\[ (\text{INSTRUCTOR} \times \text{COURSE-OFFERING}) \]
\[ \{ \text{ID-key}=\text{INSTRUCTOR-ID-in-key} \} \]
(*selection*)
All combinations of instructors and their offerings of Databases =
\[ (\text{INSTRUCTOR} \times \text{COURSE-OFFERING}) \]
\[ \{ \text{ID-key}=\text{INSTRUCTOR-ID-in-key} \} \]
\[ \{ \text{COURSE-NAME-in-key}=\text{‘Databases’} \} \]
(*selection*)
The last names of the instructors offering Databases =
\[ (\text{INSTRUCTOR} \times \text{COURSE-OFFERING}) \]
\[ \{ \text{ID-key}=\text{INSTRUCTOR-ID-in-key} \} \]
\[ \{ \text{COURSE-NAME-in-key}=\text{‘Databases’} \} \]
\[ \{ \text{LAST-NAME} \} \]
(*projection*)
6. The **join operator** combines the rows of the first table with "related" rows of the second table. It is equivalent to a selection from the cartesian product.

\[
T[\text{boolexp } (\text{attributes })]T' = (T \times T')[\text{boolexp } (\text{attributes })]
\]

**Example 4-22.**

Names of instructors teaching databases.

All combinations of instructors and their offerings =

\[
\text{(INSTRUCTOR [ID-key=INSTRUCTOR-ID-in-key] COURSE-OFFERING)}
\]

(*join*)

The last names of the instructors offering databases =

\[
\text{(INSTRUCTOR [ID-key=INSTRUCTOR-ID-in-key] COURSE-OFFERING)} \quad \text{[COURSE-NAME-in-key=‘Databases’]} \quad \text{[LAST-NAME]}
\]

7. The **natural join operator** combines two tables according to the equal values of the common attributes (column names) of the two tables.

Let the table \( T \) have \( k \) attributes with names identical to the names of \( k \) attributes of the table \( T' \), that is:

- the attributes of \( T \) are: \( A_1, \ldots, A_k, A_{k+1}, \ldots, A_n \)
- the attributes of \( T' \) are: \( A_1, \ldots, A_k, A'_1, \ldots, A'_n \)

Then

\[
T \bowtie T' = \text{get } A_1:v_1, \ldots, A_n:v_n, A'_{k+1}:w_{k+1}, \ldots, A'_n:w'_n,
\]

where

- exists \( x \) in \( T \):
  
  \( x.A_i=v_i \) and \( \cdot \cdot \cdot \) and \( x.A_n=v_n \)

- and exists \( x \) in \( T' \):
  
  \( x.A_1=v_1 \) and \( \cdot \cdot \cdot \) and \( x.A_k=v_k \) and \( x.A'_k=v'_k \) and \( \cdot \cdot \cdot \) and \( x.A'_n=w'_n \)

**Example 4-23.**

Names of instructors teaching databases.

The table \textit{INSTRUCTOR} with the column \textit{ID-key} renamed in order to be naturally

Rishe-DDS p. 4.15
Uses of the Relational Algebra:

2. Specification of queries. Albeit, the language is not friendly enough to be used for specification of complex queries.
3. An intermediate language, because it is easy to implement Relational Algebra. Other, more friendly languages, can be translated into Relational Algebra.
4. A tool to evaluate and compare different languages. We can estimate the expressive power of an arbitrary language by checking whether it is
   
a. Able to specify every query expressible in Relational Algebra.
b. Able to specify every query expressible in Relational Algebra and more.
c. Able to specify a subset of the queries expressible in Relational Algebra, where the subset is defined by weakening the Algebra through eliminating some of its operators. The list of the eliminated operators shows the weakness of the language. The list of the remaining operators shows the power of the language:
   
   - Many simple query languages can express projection and selection but not join or difference.
   - More powerful languages can express join.
   - Languages which are even more powerful can also express difference.

4.4. SQL

Rishe-DDS p. 4.16
4.4.1. Preview

SQL has become a very popular language of commercial relational database management systems.

The acronym SQL stands for Structured Query Language.

A basic query in SQL selects the values of some attributes from some rows of a table or tables where the rows satisfy a condition

Example 4-24.
When was student Russel born?

select BIRTH-YEAR
from STUDENT
where LAST-NAME='Russel'

Example 4-25.
List the names of all students.

select FIRST-NAME, LAST-NAME
from STUDENT
where true

Example 4-26.
What courses has Prof. Graham taught?

select COURSE-NAME
from COURSE-OFFERING, INSTRUCTOR
where INSTRUCTOR.NAME = 'Graham' and INSTRUCTOR.ID = COURSE-OFFERING. INSTRUCTOR-ID

4.4.2. Basic queries

Syntax:

select expression₁, . . . , expressionₙ
from table₁ var₁, . . . , tableₙ varₙ

Rishe-DDS p. 4.17
**where condition**

The *condition* is a Boolean expression without quantifiers. It may depend on the variables \( \text{var}_1, \ldots, \text{var}_n \).

**Meaning:**

**get** \( \text{expression}_1, \ldots, \text{expression}_n \)

**where**

\( \text{var}_1 \text{ is a table}_1 \text{ and } \cdots \text{ var}_n \text{ is a table}_n \text{ and } \text{condition} \)

**Example 4-27.**

Print the names of the pairs of students who live together.

```sql
select s_1.LAST-NAME, s_2.LAST-NAME
from STUDENT s_1, STUDENT s_2
where s_1.ADDRESS = s_2.ADDRESS
```

**Abbreviation:**

If a table \( T_i \) appears exactly once in the *from* list, then it does not have to be explicitly accompanied by a variable. Implicitly, the name of the variable is identical to the name of the table.

```sql
select \text{expression}_1, \ldots, \text{expression}_n
from \text{table}_1, \ldots, \text{table}_n
where \text{condition}
```

**Example 4-28.**

Print the names of the pairs of a student and an instructor who live together. (This includes an instructor who is also a student and lives with alone.)

```sql
select STUDENT.LAST-NAME, INSTRUCTOR.LAST-NAME
from STUDENT, INSTRUCTOR
where STUDENT.ADDRESS = INSTRUCTOR.ADDRESS
```

**Abbreviation:** When there is only one table in the *from* list, then whenever \( T\text{.attribute} \) appears in the query, it may be shortened to *attribute* without the prefix \( T\).
Example 4-29.
Print the names and the addresses of all computer science students.

\[
\text{select } \text{LAST-NAME, ADDRESS} \\
\text{from STUDENT} \\
\text{where MAJOR-DEPARTMENT-MAIN-NAME = ‘Computer Science’}
\]

Abbreviation: When the \textit{select} list consists of all the attributes of the \textit{from} tables, the \textit{select} list may be abbreviated by \textit{"select \*".}

Example 4-30.
Print the names (last and first), the IDs, the birth-years, the major and minor departments, and the addresses of all computer science students.

\[
\text{select \*} \\
\text{from STUDENT} \\
\text{where MAJOR-DEPARTMENT-MAIN-NAME = ‘Computer Science’}
\]

Note:
- The output of a query is a partial instantaneous binary database. It can be printed as a table (in the common sense of the word \textit{table}).
- Often, but not always, the output of a query is an instantaneous table in the sense of the Relational Model. This is not always true since the output of a query may contain identical rows, while a relational instantaneous table may not contain identical rows.

4.4.3. Basic aggregates
Basic aggregates are predefined functions which are applied to the whole output of a query. Syntactically, the functions are applied to the expression(s) in the \textit{select} list.

1. \textbf{count} — when this function is applied to the \textit{select} list, it replaces the output of the query by the number of rows in the output.

Example 4-31.
How many computer science students are there?

\[
\text{select count(\*)} \\
\text{from STUDENT} \\
\text{where MAJOR-DEPARTMENT-MAIN-NAME = ‘Computer Science’}
\]
2. **avg** — when the *select* list consists of only one expression, and the expression produces numerical values, the function **avg** replaces the output by the average of the values in the output.

   **Example 4-32.**
   What is the average grade in the *Databases* course?
   ```sql
   select avg(FINAL GRADE)
   from COURSE-ENROLLMENT
   where COURSE-NAME-IN-KEY = 'Databases'
   ```

3. **sum** — when the *select* list consists of only one expression, and the expression produces numerical values, the function **sum** replaces the output by the sum of the values in the output.

4. **max** — when the *select* list consists of only one expression, and the expression produces numerical values, the function **max** replaces the output by the maximum of the values in the output.

5. **min** — when the *select* list consists of only one expression, and the expression produces numerical values, the function **min** replaces the output by the minimum of the values in the output.

6. **distinct** — eliminates duplicate rows in the output. This function must be applied to the whole *select* list. (In many implementations, this function is called "**unique.**")

   **Example 4-33.**
   List the distinct addresses of the students. (Do not list the same address twice.)
   ```sql
   select distinct ADDRESS
   from STUDENT
   where TRUE
   ```

7. The function **distinct** can be combined with any other aggregate function. The function **distinct** is applied first, and then another function is applied to the result.

   **Example 4-34.**
   How many departments have minor students?
   ```sql
   select count (distinct MINOR-DEPARTMENT-MAIN-NAME)
   from STUDENT
   where TRUE
   ```

Rishe-DDS p. 4.20
4.4.4. Nested queries

Query forms are represented in SQL by allowing some expressions to contain variables which are not defined (either explicitly or implicitly) in the from list. A query form would become a query if the expressions with undefined variables were replaced by constants. Query forms are used in SQL primarily in order to construct nested queries.

Example 4-35.

The following is not a query because it contains an undefined variable s. It is a query form, which would become a query if the expression s.ID-KEY were replaced by a constant, such as 345466.

```sql
select *
from COURSE-ENROLLMENT
where s>ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY
```

The nested queries are obtained in SQL by extending the syntax of the where condition by allowing the following subconditions within the condition:

1. exists query-form

   This subcondition gives TRUE when the result of the query form is not empty — when it contains at least one row. (This subcondition is evaluated when all the variables on which the query form depends are interpreted.)

Example 4-36.

Find the names of the students who never took a course.

```sql
select LAST-NAME
from STUDENT
where
    not exists
    (select *
     from COURSE-ENROLLMENT
     where STUDENT.ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY)
```

Example 4-37.

List the instructor IDs and course names such that the instructor is the exclusive teacher of the course (i.e. no other instructors have offered the course).

```sql
select INSTRUCTOR-ID-in-key, COURSE-NAME-in-key
from COURSE-OFFERING co
```
where not exists

select INSTRUCTOR-ID-in-key
from COURSE-OFFERING co1
where co.COURSE-NAME-in-key=co1.COURSE-NAME-in-key and
not co.INSTRUCTOR-ID-in-key=co1.INSTRUCTOR-ID-in-key

2. expression in query-form-producing-only-one-value-per-row
This subcondition gives TRUE when the value of the expression constitutes a row in the output of the query-form

Example 4-38.
Find the names of the students who took at least one course.

select LAST-NAME
from STUDENT
where

(ID KEY in

select STUDENT-ID-IN-KEY
from COURSE-ENROLLMENT
where TRUE)

3. expression not in query-form-producing-only-one-value-per-row
This subcondition gives TRUE when the value of the expression does not constitute a row in the output of the query-form

Example 4-39.
Find the names of the students who never took a course.

select LAST-NAME
from STUDENT
where

(ID KEY not in

select STUDENT-ID-IN-KEY
from COURSE-ENROLLMENT
where TRUE)
4. \(<\text{expressions}>\) \textbf{in} \ query-form

This subcondition gives TRUE when the values of the \textit{expressions} constitute a row in the output of the \textit{query-form}

\textbf{Example 4-40.}

Find the names of the students who may be spouses of instructors — those who have the same last name and address as an instructor.

\begin{verbatim}
select LAST-NAME, FIRST-NAME 
from STUDENT 
where 
<LAST-NAME, ADDRESS> in 
    select LAST-NAME, ADDRESS 
from INSTRUCTOR 
where TRUE 
\end{verbatim}

5. \(<\text{expressions}>\) \textbf{not in} \ query-form

This subcondition gives TRUE when the values of the \textit{expressions} do not constitute a row in the output of the \textit{query-form}

\textbf{Example 4-41.}

Find the names of some students who are certainly not spouses of instructors.

\begin{verbatim}
select LAST-NAME, FIRST-NAME 
from STUDENT 
where 
<LAST-NAME, ADDRESS> not in 
    select LAST-NAME, ADDRESS 
from INSTRUCTOR 
where TRUE 
\end{verbatim}

6. \textit{query-form \textbf{contains} \ query-form}

This subcondition gives TRUE when every row produced by the right \textit{query form} is also produced by the left \textit{query form}.

\textbf{Example 4-42.}

Find the names of the students who took all the courses.

\begin{verbatim}
select LAST-NAME, FIRST-NAME 
\end{verbatim}
from STUDENT
where
  select COURSE-NAME-IN-KEY
from COURSE-ENROLLMENT
where STUDENT.ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY
contains
  select NAME-KEY
from COURSE
where TRUE

7. expression comparison query-form-producing-only-one-value
The allowed comparisons are: =, <, >, >=, <=, <>.

Example 4-43.
Find the names of the students who took more than 1000 course offerings.

  select LAST-NAME, FIRST-NAME
  from STUDENT
where
  1000 <

  select count (*)
  from COURSE-ENROLLMENT
where STUDENT.ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY

8. expression comparison any query-form
This subcondition is TRUE if the comparison with at least one row of the query-form’s output is TRUE.

Example 4-44.
Find the names of the students who studied something in the first 20 calendar years of their life.

  select LAST-NAME, FIRST-NAME
  from STUDENT
where
9. expression comparison all query-form

This subcondition is TRUE if the comparison with every one of the query-form’s output rows is TRUE.

Example 4-45.
Find the names of the students who studied nothing (as far as the database knows) in the first 20 calendar years of their life.

```sql
select LAST-NAME, FIRST-NAME
from STUDENT
where
BIRTH-YEAR + 20 <= all
```

```sql
select YEAR
from COURSE-ENROLLMENT
where STUDENT.ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY
```

Example 4-46.
List the instructor IDs and course names such that the instructor is the exclusive teacher of the course (i.e. no other instructors have offered the course).

```sql
select INSTRUCTOR-ID-in-key, COURSE-NAME-in-key
from COURSE-OFFERING co
where INSTRUCTOR-ID-in-key = all
```

```sql
select INSTRUCTOR-ID-in-key
from COURSE-OFFERING co1
where co.COURSE-NAME-in-key=co1.COURSE-NAME-in-key
```
Example 4-47.

An alternative code for the above query:

```
select INSTRUCTOR-ID-in-key, COURSE-NAME-in-key
from COURSE-OFFERING co
where 1 =
    select count (distinct INSTRUCTOR-ID-in-key)
from COURSE-OFFERING co1
where co.COURSE-NAME-in-key=co1.COURSE-NAME-in-key
```

4.4.5. Grouping of rows

The aggregate functions can be applied to subsets of rows produced by `select`. For this purpose, the rows resulting from `select` can be partitioned into groups according to the values of some attributes.

Syntax:

```
select expression_1, . . . , expression_n
from table_1 var_1, . . . , table_n var_n
where condition
group by attribute_1, . . . , attribute_k
```

Each `attribute_i` has the form

```
variable.attribute-name
```

When no ambiguity arises, the table-name can be used instead of the variable:

```
table-name.attribute-name
```

When no ambiguity further arises, the table-name can be omitted:

```
attribute-name
```

Meaning:

The rows satisfying the `condition` are combined into groups so that in each group the attributes of the grouping have constant values; that is, two rows `r_1` and `r_2` are in the same group if and only if

```
r_1.attribute_1 = r_2.attribute_1 and
r_1.attribute_2 = r_2.attribute_2 and . . .
r_1.attribute_k = r_2.attribute_k
```

For every group, only one cumulative row is produced in the result. The resulting cumulative row is obtained by evaluation of the `expressions` of the `select` clause. The aggregates in those expressions are interpreted as applying not to the whole output but
only to the rows comprising one group.

**Example 4-48.**
For every department, list the number of instructors it employs.

```sql
select DEPARTMENT-MAIN-NAME-in-key, count(INSTRUCTOR-ID-in-key)
from WORK
group by DEPARTMENT-MAIN-NAME-in-key
```

**Example 4-49.**
For every student who took classes in a summer, for every instructor who gave grades to the student in a summer, print the average of the summer grades given by the instructor to the student.

```sql
select STUDENT-ID-in-key, INSTRUCTOR-ID-in-key, avg(FINAL-GRADE)
from COURSE-ENROLLMENT
where SEASON = 'Summer'
group by STUDENT-ID-in-key, INSTRUCTOR-ID-in-key
```

Some of the groups produced by `group by` can be screened out according to the values of aggregate functions applied to the group. A group screening condition can be specified in a `having` clause as follows.

```sql
select expressions
from tables
where condition-on-the-source-rows-of-the-tables
group by attributes
having condition-on-the-groups
```

The `condition-on-the-groups` is a Boolean expression. The aggregate functions appearing in this condition apply to the rows comprising one group.

**Example 4-50.**
What departments employ more than 100 instructors each?

```sql
select DEPARTMENT-MAIN-NAME-in-key
from WORK
where true
group by DEPARTMENT-MAIN-NAME-in-key
```
Example 4-51.
For every student who took classes in a summer, for every instructor who gave grades to the student in a summer, so that the average of the summer grades given by the instructor to the student is greater than 60, print the average of the summer grades given by the instructor to the student.

\[
\text{select } \text{STUDENT-ID-in-key, INSTRUCTOR-ID-in-key, } \text{avg(FINAL-GRADE)} \\
\text{from } \text{COURSE-ENROLLMENT} \\
\text{where } \text{SEASON} = \text{‘Summer’} \\
\text{group by } \text{STUDENT-ID-in-key, INSTRUCTOR-ID-in-key} \\
\text{having } \text{avg(FINAL-GRADE)} > 60
\]

Example 4-52.
What students have taken classes with every instructor?

\[
\text{select } \text{STUDENT-ID-in-key} \\
\text{from } \text{COURSE-ENROLLMENT} \\
\text{group by } \text{STUDENT-ID-in-key} \\
\text{having } \text{count(}x\text{)} = \\
\quad (\text{select count(}x\text{) from INSTRUCTOR})
\]

4.4.6. Sorting
The output of an SQL query can be sorted for the purpose of printing in any desired order or for delivery to an application program in a desired order. This is accomplished by an \textbf{order by} clause specifying one or more attributes to sort by:

\[
\text{query} \\
\text{order by } \text{attributes}
\]
When more than one sorting attribute is given, the output’s primary order is according to the first attribute, then according to the second, and so on.

Example 4-53.
List all computer science majoring students sorted by their minors.

\[
\text{select } * \\
\text{from } \text{STUDENT}
\]
where MAJOR-DEPARTMENT-MAIN-NAME = ‘Computer Science’
order by MINOR-DEPARTMENT-MAIN-NAME

4.4.7. Update transactions

Update transactions can be specified in SQL.
1. Deleting a set of rows from a table.

   delete from table
   where condition-on-rows-to-be-deleted

   Example 4-54.
   Delete the student whose ID is 11111.
   delete from STUDENT
   where ID-key = 11111

   Example 4-55.
   Delete all music majors.
   delete from STUDENT
   where MAJOR-DEPARTMENT-MAIN-NAME = ‘Music’

2. Inserting a row into a table.

   insert into table
   attribute_1, . . . , attribute_n
   values
   value_1, . . . , value_n

   The attributes of the table which are not specified in the insert command are set to null values for the row being inserted.

   Example 4-56.
   Let the instructor whose ID is 22222 work in the department whose main name is Arts.
   insert into WORK
   INSTRUCTOR-ID-in-key, DEPARTMENT-MAIN-NAME-in-key
   values
3. Inserting a set of rows into a table. A set of rows to be inserted can be defined as the result of a query.

\[
\text{insert into } table \\
\text{attribute}_1, \ldots, \text{attribute}_n \\
\text{query}
\]

**Example 4-57.**
Let all physics instructors work also in Arts.

\[
\text{insert into } WORK \\
\text{INSTRUCTOR-ID-in-key, DEPARTMENT-MAIN-NAME-in-key} \\
\text{select INSTRUCTOR-ID-in-key, `Arts'} \\
\text{from WORK} \\
\text{where DEPARTMENT-MAIN-NAME-in-key = `Physics'}
\]

4. Modifying the values of some attributes in a set of rows of a table. The set of rows can be specified by a \text{where} condition. The new values can be specified as constants or as expressions using the old values of the row being updated.

\[
\text{update } table \\
\text{set} \\
\text{attribute}_1=\text{expression}_1, \ldots, \text{attribute}_n=\text{expression}_n \\
\text{where condition}
\]

**Example 4-58.**
Decrease by 10 percent all grades above 90.

\[
\text{update } COURSE-ENROLLMENT \\
\text{set FINAL-GRADE = FINAL-GRADE*0.9} \\
\text{where FINAL-GRADE > 90}
\]
4.4.8. DDL

SQL has a data definition capability.

Specification of a table

```
create table table-name
    attribute_1 data-type_1, ..., attribute_n data-type_n
```

Example 4-59.

```
create table QUARTER
    YEAR-in-key Integer
    SEASON-in-key String
```

Specification of a userview table

```
create view new-table-name
    attribute_1, ..., attribute_n

as select-command
```

Example 4-60.

```
create view TAUGHT
    STUDENT-ID HIS-TEACHER-ID

as select STUDENT-ID-in-key, INSTRUCTOR-ID-in-key

from COURSE-ENROLLMENT
```

4.4.9. SQL extension of Pascal

SQL can be used not only interactively but also as a DML extension of a programming language. This section shows how SQL can be embedded in Pascal. The embedding in other programming languages is similar.

Host variables in SQL statements

Wherever a constant can appear in SQL, a host program variable can appear instead. Before the SQL statement is performed, the variable is evaluated to give a value. To distinguish between host program variables and SQL variables, the host variables are preceded by a colon (:).

Example 4-61.

Create a new course whose name is in the Pascal variable `course-name`.

```
insert into COURSE
```

Rishe-DDS p. 4.31
Example 4-62.
For each standard input line create a new course whose name is the string appearing in that line.

```plaintext
var course-name: String;
begin
  while not eof(Input) do begin
    readln (course-name);
    insert into COURSE
    NAME-key values :course-name
  end
end.
```

Retrieving a row of values from the database

If we anticipate that a select command will retrieve exactly one row of data, we can have this data placed into variables of the host program by the following command.

```
select-command into host-variables
```

Example 4-63.

(* Get the total number students born in 1980 into the variable myvar. *)

```plaintext
select count (*)
from STUDENT
where BIRTH-YEAR = 1980
into :myvar
```

Processing of multirow output of a query

The program can retrieve a set of rows of data from the database.

- Such a set of rows can be defined as the output of a select command. The program would then process the retrieved rows, one row at a time.

Example 4-64.

```plaintext
select BIRTH-YEAR
```
from STUDENT
where MAJOR-DEPARTMENT-MAIN-NAME = ‘Management’;

- To scan such rows in a program, SQL defines a cursor, a logical pointer to the current row. The declaration of a cursor defines a query.

Example 4-65.

```sql
declare current-student cursor for
    select BIRTH-YEAR
    from STUDENT
    where MAJOR-DEPARTMENT-MAIN-NAME = ‘Management’;
```

- The opening of the cursor performs the query.

Example 4-66.

```sql
open current-student;
```

- The fetch command then brings to the program one row each time and advances the cursor to point to the next row.

Example 4-67.

```sql
fetch current-student into :birth-year;
```

- When a fetch is attempted beyond the last row in the output of a query, the special variable `sqlstatus` is set to the value of the special constant `not-found`.

Example 4-68.

```sql
(* Print the logarithm of the birth year of every management student (major) *)

declare current-student cursor for
    select BIRTH-YEAR
    from STUDENT
    where MAJOR-DEPARTMENT-MAIN-NAME = ‘Management’;
open current-student;
repeat
    fetch current-student into :birth-year;
```
if sqlstatus ≠ not-found then writeln(log(birth-year))
until sqlstatus = not-found

The relevant commands are:

a. declare cursor-name cursor for select-command
b. open cursor-name
c. fetch cursor-name into host-variables

Update and delete of fetched rows

After a row has been fetched, it can be updated or deleted provided the row belongs to one table of the schema (and not to a join of tables).

a. update table-name
   set
   attribute\_1 = expression\_1, \ldots, attribute\_n = expression\_n
   where current of cursor-name

Example 4-69.

(* Replace the birth year of every management student (major) by the logarithm of the birth year. *)

declare current-student cursor for
    select BIRTH-YEAR
    from STUDENT
    where MAJOR-DEPARTMENT-MAIN-NAME = ‘Management’;
open current-student;
repeat
    fetch current-student into :birth-year;
    birth-year := log (birth-year);
    if sqlstatus ≠ not-found then
        update STUDENT
        set BIRTH-YEAR = :birth-year
        where current of current-student
    until sqlstatus = not-found
b. \texttt{delete from table-name}
\texttt{where current of cursor-name}

\begin{example}
(*) Display the name of every student; prompt the user whether the student should be deleted; if the user replies ‘yes’ — delete the student. *)

\texttt{declare current-student cursor for}
\begin{verbatim}
select LAST-NAME, FIRST-NAME
from STUDENT
\end{verbatim}
\texttt{open current-student;}
\texttt{repeat}
\begin{verbatim}
fetch current-student into :last-name, :first-name;
if sqlstatus ≠ not-found then begin
  writeln (‘ Would you like to delete’, last-name, ‘’, first-name, ‘?’);
  readln (answer);
  if answer=’yes’ then
    delete from STUDENT
    where current of current-student
  end
\end{verbatim}
\texttt{until sqlstatus = not-found}
\end{example}

4.5. \textbf{Expressive Power of Relational Query Languages}

Ignoring minor differences in expressiveness, such as the output of identical rows, the following languages have approximately equal power. A query that can be expressed in one language can also be expressed in the others:

- Tuple-oriented Relational Calculus without aggregate functions
- Relational Algebra
- SQL without aggregate functions

The aggregate extension of the Predicate Calculus and SQL with aggregate functions have a higher power.

The structured extension of Pascal, being a general-purpose programming language, has a much higher power of expressiveness.

Rishe-DDS p. 4.35
Reference Schemas

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.

![Semantic Schema](image)

**Figure Ref-1.** A semantic schema for a university application.

Rishe-DDS p. 4.36
<table>
<thead>
<tr>
<th>TABLE</th>
<th>PRIMARY KEY</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUDENT</td>
<td>id-key: Integer 1:1</td>
<td>last-name: String, first-name: String, birth-year: 1870..1990, address: String, major-department-main-name: String, minor-department-main-name: String</td>
</tr>
<tr>
<td>DEPARTMENT</td>
<td>main-name-key: String 1:1</td>
<td></td>
</tr>
<tr>
<td>COURSE ENROLLMENT</td>
<td>instructor-id-in-key: Integer</td>
<td>course-name-in-key: String, year-in-key: 1980..1995, season-in-key: String, student-id-in-key: Integer, final-grade: 0..100</td>
</tr>
<tr>
<td>INSTRUCTOR</td>
<td>id-key: Integer 1:1</td>
<td>last-name: String, first-name: String, birth-year: 1870..1990, address: String</td>
</tr>
<tr>
<td>DEPARTMENT</td>
<td>main-name-key: String 1:1</td>
<td></td>
</tr>
<tr>
<td>WORK</td>
<td>instructor-id-in-key: Integer</td>
<td>department-main-name-in-key: String</td>
</tr>
<tr>
<td>COURSE</td>
<td>name-key: String 1:1</td>
<td></td>
</tr>
<tr>
<td>QUARTER</td>
<td>year-in-key: 1980..1995</td>
<td>season-in-key: String</td>
</tr>
</tbody>
</table>

**Figure Ref-2.** A relational schema for the university application.

Rishe-DDS p. 4.37
5. MORE ON DATABASE DESIGN METHODOLOGY

This chapter begins with a case study of a design of an actual database application. Section 2 summarizes the flow of database design. Section 3 compares the methodology of this book to the older methodology of normalization.

5.1. A Case Study of an Application

The university database studied in the examples of the other chapters of this book is a small “toy” application. The objective of this chapter is to show a real application. This chapter describes a database that has been developed for the Hydrology Division of the Everglades National Park. (Actually this application is a self-contained subapplication of a larger database covering various activities of the Park and consisting of more than 1000 categories, relations, and attributes.)

The first part of this chapter presents the semantic analysis of the relevant subschemas. It contains information based on our interviews with the Client’s representatives. We have translated this analysis into formal concepts. The specification of every concept herein consists of:

- The concept’s name, which should be clear and meaningful to the database users
- Technical characteristics of the concept
- A comment defining the meaning of the concept

A correct definition in the comment is important. Its purposes are to:

- Verify that the systems analysts correctly understand the meanings of the application’s concepts
- Concisely convey the meanings of the application to the programming personnel who will work on the application in the future
- Provide online comments on all database entities to the future users of the database on the Client’s side
- Provide an information reference manual for use by the Client’s personnel and for training of new employees, whether they will be using the database or not
- Facilitate decision making at the Client’s managerial and executive levels by providing a graphic overview and a comprehensive directory of the information owned by the Client (as a supplement to the other decision support resources: a directory of the personnel employed, a directory of financial and tangible assets owned, and the database itself)
- Specify informally integrity constraints beyond those shown in the graphical schema

For every numeric attribute, a range of its possible values is given. For example, 23.5..100.7 means that the values may not be less than 23.5 or greater than 100.7 and that the precision is
one digit after the decimal point. It is desirable that a range be as narrow as possible while still allowing for all the possible values that may be meaningful in the database at any time in the future. The range specification is used to check the input in order to eliminate data entry typos.

The schema is partitioned into several subschemas each of which is small enough to be displayed on one page. The interconnections between the subschemas are shown by having some categories appear in more than one subschema. For every category, one subschema is the **home subschema** of that category and contains all of its attributes. If that category appears also in other subschemas, it has no attributes there, but, instead, a reference to its home subschema. The category appears in subschemas other than its home subschema in order to show relations with other categories of those subschemas.

### Example 5-1.

The Equipment History subschema covers information relevant to all the equipment of the Park. It is the home subschema of the category **EQUIPMENT**. The Hydrology Stations Equipment subschema has additional information about the equipment installed at hydrological stations. The category **EQUIPMENT** appears in Figure 5-1 as a pointer to the Equipment History subschema.

![Figure 5-1. Interschema reference.](image)

### 5.1.1. Semantic analysis

The semantic categories of this section and the corresponding relational tables are listed in the Index of this book. This facilitates cross-referencing between them.

#### 5.1.1.1. Hydrology stations (includes marine rainfall stations)

Principal interviewer: Michael Alexopoulos. Client representatives interviewed: De Witt Smith, David Sikema. Revised by Naphtali Rishe 03/19/91.

- **HYDROLOGY-STATION** — category (A catalog of hydrology stations which reside within the Everglades National Park.)
- **FIXED-STATION** — subcategory of **HYDROLOGY-STATION** (A hydrology station which is housed in a permanent structure.)
- **TEMPORARY-STATION** — subcategory of **HYDROLOGY-STATION** (A hydrology station which only exists for a period of time and it is not housed in a permanent structure.)
**HYDROLOGY STATION**

- no-other
- station-id: Char(15) key
- station-description: String
- station-location-north: 2746840..2865840
- station-location-east: 446880..563280
- method-used: 'standard-survey', 'loran', 'gps-equipment', 'map'
- location-tolerance-ft: 0..1000
- station-owner: 'everglades-national-park', 'usgs', 'sfwmd', 'us-coe'

**FIXED STATION**

- disjoint: FIX/TEM/
- no-other
- housing-descriptor: String
- platform-height-ft: 0.00..10.00
- benchmark-location-north: 2746840..2865840
- benchmark-location-east: 446880..563280
- platform-benchmark-height-difference: 0.00..10.00

**TEMPORARY STATION**

- begin-date: Date
- end-date: Date

**DISCONTINUOUS STATION**

- disjoint: CON/DIS/

**STATION CONTINUITY PERIOD**

- begin-date: Date  key/2
- end-date: Date

**Figure 5-2.** Semantic subschema for Hydrology Stations.

- **CONTINUOUS-STATION** — subcategory of **FIXED-STATION** (A fixed hydrology station which collects data continuously.)

- **DISCONTINUOUS-STATION** — subcategory of **FIXED-STATION** (A fixed hydrology station which collects data only for specific intervals of time.)

Rishe-DDS p. 5.3
Every object of the category \textit{HYDROLOGY-STATION} must also belong to its subcategory. The following subcategories are disjoint: FIXED-STATION TEMPORARY-STATION.

Every object of the category \textit{FIXED-STATION} must also belong to its subcategory. The following subcategories are disjoint: CONTINUOUS-STATION DISCONTINUOUS-STATION.

\begin{itemize}
  \item \textit{the-discontinuous-station} -- relation from \textit{STATION-CONTINUITY-PERIOD} to \textit{DISCONTINUOUS-STATION} (key/2) (The discontinuous station which was active for periods of time collecting data.)
\end{itemize}

The objects of the category \textit{HYDROLOGY-STATION} are identified by: station-id.

The objects of the category \textit{STATION-CONTINUITY-PERIOD} are identified by: begin-date the-discontinuous-station.

\begin{itemize}
  \item \textit{station-id} -- attribute of \textit{HYDROLOGY-STATION}, range: \textit{Char(15)} (key) (Identification.)
  \item \textit{station-description} -- attribute of \textit{HYDROLOGY-STATION}, range: \textit{String (m:1)} (English name or designation of the station.)
  \item \textit{station-location-north} -- attribute of \textit{HYDROLOGY-STATION}, range: 2746840..2865840 (m:1) (UTM north coordinate of a hydrology station.)
  \item \textit{station-location-east} -- attribute of \textit{HYDROLOGY-STATION}, range: 446880..563280 (m:1) (UTM east coordinate of a hydrology station.)
  \item \textit{method-used} -- attribute of \textit{HYDROLOGY-STATION}, range: \textit{standard-survey, loran, gps-equipment, map} (m:1) (The method used to derive the location coordinates of a station.)
  \item \textit{location-tolerance-ft} -- attribute of \textit{HYDROLOGY-STATION}, range: 0..1000 (m:1) (Tolerance of the location of a station, in feet. A value \(x\) assigned to this attribute means that the tolerance is +/-\(x\) feet.)
  \item \textit{station-owner} -- attribute of \textit{HYDROLOGY-STATION}, range: \textit{everglades-national-park, usgs, sfwm, us-coe} (m:1) (The agency which owns the station.)
  \item \textit{housing-descriptor} -- attribute of \textit{FIXED-STATION}, range: \textit{String (m:1)} (Description of the housing of a fixed station.)
  \item \textit{platform-height-ft} -- attribute of \textit{FIXED-STATION}, range: 0.00..10.00 (m:1) (The height of the station platform from the water surface, in feet.)
  \item \textit{benchmark-location-north} -- attribute of \textit{FIXED-STATION}, range: 2746840..2865840 (m:1) (UTM north coordinate of the benchmark which corresponds to a fixed station.)
  \item \textit{benchmark-location-east} -- attribute of \textit{FIXED-STATION}, range: 446880..563280 (m:1) (UTM east coordinate of the benchmark which corresponds to a fixed station.)
\end{itemize}

Rishe-DDS p. 5.4
platform-benchmark-height-difference — attribute of FIXED-STATION, range: 0.00..10.00 (m:1) (The difference between the height of the station platform and the height of its corresponding benchmark, in feet.)

begin-date — attribute of STATION-CONTINUITY-PERIOD, range: Date (key/2) (The date during which a discontinuous station was activated and started the generation of data for some parameters.)

date-end — attribute of STATION-CONTINUITY-PERIOD, range: Date (m:1) (The date during which a period of activation for some discontinuous station ended.)

begin-date — attribute of TEMPORARY-STATION, range: Date (m:1) (The starting date of the life of a temporary station.)

date-end — attribute of TEMPORARY-STATION, range: Date (m:1) (The ending date of the life of a temporary station.)

5.1.1.2. Equipment

Principal interviewer: Michael Alexopoulos. Client representatives interviewed: De Witt Smith, David Sikema. Revised by Naphtali Rishe 03/19/91.

EQUIPMENT — category (A general catalog of equipment owned by the Everglades National Park.)

REPAIR-PERIOD — category (A list of periods during which various repairs were made to equipment.)

BATTERY-CHANGE — category (A list of dates during which a battery change was done to equipment.)

CALIBRATION — category (A list of calibrations done to equipment.)

CALIBRATION-USING-EQUIPMENT — subcategory of CALIBRATION (Calibration of equipment while using other equipment.)

CALIBRATION-USING-OTHER-TECHNIQUE — subcategory of CALIBRATION (Calibration using a technique which is documented here. However this technique does not use instruments which are recorded in the database, i.e., this category is disjoint from CALIBRATION-USING-EQUIPMENT.)

Every object of the category CALIBRATION must also belong to its subcategory. The following subcategories are disjoint: CALIBRATION-USING-EQUIPMENT CALIBRATION-USING-OTHER-TECHNIQUE.

the-equipment-repaired — relation from REPAIR-PERIOD to EQUIPMENT (key/2) (The equipment repaired.)

in-equipment — relation from BATTERY-CHANGE to EQUIPMENT (key/2) (The equipment for which the battery was changed.)

the-equipment-calibrated — relation from CALIBRATION to EQUIPMENT (key/2) (The equipment which was calibrated.)

Rishe-DDS p. 5.5
The objects of the category \textit{CALIBRATION} are identified by: calibration-date the-equipment-calibrated.

The objects of the category \textit{REPAIR-PERIOD} are identified by: begin-date the-equipment-repaired.

The objects of the category \textit{BATTERY-CHANGE} are identified by: battery-change-date in-equipment.

The objects of the category \textit{EQUIPMENT} are identified by: park-service-number.

---

\textit{has-used} — relation from \textit{CALIBRATION-USING-EQUIPMENT} to \textit{EQUIPMENT} \((m:m)\) (The equipment used to calibrate some other equipment.)
5.1.1.3. Daily hydrology observations

Principal interviewer: Michael Alexopoulos. Client representative interviewed: David Sikema. Revised by Naphtali Rishe 03/19/91.

- **DAILY-HYDROLOGY-OBSERVATION** — category (A catalog of daily hydrology observations which originate from stations within the Everglades National Park.)
- **DAILY-STAGE** — subcategory of **DAILY-HYDROLOGY-OBSERVATION** (Daily mean stage measurements.)
- **DAILY-RAINFALL** — subcategory of **DAILY-HYDROLOGY-OBSERVATION** (Daily total rainfall measurements.)
- **DAILY-DISCHARGE** — subcategory of **DAILY-HYDROLOGY-OBSERVATION** (Daily mean discharge measurements.)
- **DAILY-TEMPERATURE** — subcategory of **DAILY-HYDROLOGY-OBSERVATION** (Daily temperature measurements.)
Every object of the category *DAILY-HYDROLOGY-OBSERVATION* must also belong to its subcategory. The following subcategories are disjoint: *DAILY-EVAPORATION* *DAILY-TEMPERATURE* *DAILY-DISCHARGE* *DAILY-RAINFALL* *DAILY-STAGE*. 

Rishe-DDS p. 5.8
- HYDROLOGY-STATION —category (See subschema hydrology-stations.)

- daily-produced-by —relation from DAILY-HYDROLOGY-OBSERVATION to HYDROLOGY-STATION (key/2) (The station which generates daily measurements. Daily stage and rainfall measurements can only be generated by continuous stations owned by one of the following agencies: U.S. Army COE, SFWMD, USGS. Daily discharge measurements can only be generated by one of the following agencies: USGS, SFWMD. Daily evaporation and temperature measurements can only be generated by stations owned by one of the following agencies: Everglades RC, SFWMD, U.S. Army COE.)

- date —attribute of DAILY-HYDROLOGY-OBSERVATION, range: Date (key/2) (The date during which a hydrology observation was made.)

The objects of the category DAILY-HYDROLOGY-OBSERVATION are identified by: date daily-produced-by.

- daily-mean-stage —attribute of DAILY-STAGE, range: -99.99..+99.99 (m:1) (The daily mean stage quantity measured in ft/100. That is the value 1.23 means 0.0123 feet. This field is left blank when data is not available.)

- daily-precipitation-inch —attribute of DAILY-RAINFALL, range: 0.00..22.00 (m:1) (The daily total precipitation quantity measured in inches. This field is left blank when data is not available.)

- total-daily-evaporation-inch —attribute of DAILY-EVAPORATION, range: 0.00..12.00 (m:1) (The total water evaporation for the day, measured in inches. This field is left blank when data is not available.)

- daily-mean-discharge —attribute of DAILY-DISCHARGE, range: -99999.99..+99999.99 (m:1) (The mean discharge quantity for the day, measured in cubic feet per second. This field is left blank when data is not available.)

- formula —attribute of DAILY-DISCHARGE, range: ‘weir’, ‘rating-curves’ (m:1) (The formula used to compute the mean discharge value.)

- daily-upstream-stage-ft —attribute of DAILY-DISCHARGE, range: 0..12 (m:1) (Upstream stage level for the day, measured in feet.)

- daily-downstream-stage-ft —attribute of DAILY-DISCHARGE, range: 0..12 (m:1) (Downstream stage level for the day, measured in feet.)

- daily-mean-temperature —attribute of DAILY-TEMPERATURE, range: 20.00..120.00 (m:1) (The mean temperature for the day, in degrees Fahrenheit. This field is left blank when data is not available.)

- daily-max-temperature —attribute of DAILY-TEMPERATURE, range: 20.00..120.00 (m:1) (The maximum temperature for the day, in degrees Fahrenheit. This field is left blank when data is not available.)

- daily-min-temperature —attribute of DAILY-TEMPERATURE, range: 20.00..120.00 (m:1) (The minimum temperature for the day, in degrees Fahrenheit. This field is left blank when data is not available.)
5.1.1.4. Hourly hydrology observations

Principal interviewer: Michael Alexopoulos. Client representative interviewed: David Sikema. Revised by Naphtali Rishe 03/19/91.

**Figure 5-5.** Semantic subschema for hourly stage and rainfall observations.

- **HOURLY-HYDROLOGY-OBSERVATION** — category (A catalog of hourly hydrology observations which originate from stations within the Everglades National Park.)
- **HOURLY-STAGE** — subcategory of **HOURLY-HYDROLOGY-OBSERVATION** (Hourly mean stage measurements.)
- **HOURLY-RAINFALL** — subcategory of **HOURLY-HYDROLOGY-OBSERVATION** (Hourly total rainfall measurements.)
- **HOURLY-WIND** — subcategory of **HOURLY-HYDROLOGY-OBSERVATION** (Hourly wind speed and wind direction measurements.)

Every object of the category **HOURLY-HYDROLOGY-OBSERVATION** must also belong to its subcategory. The following subcategories are disjoint: **HOURLY-WIND** **HOURLY-
RAINFALL HOURLY-STAGE.

- **HYDROLOGY-STATION** — category (See subschema hydrology-stations.)
- **hourly-produced-by** — relation from **HOURLY-HYDROLOGY-OBSERVATION** to **HYDROLOGY-STATION** (key/3) (The station which generates hourly either stage or rainfall or wind speed and direction measurements. Hourly stage and rainfall measurements can only be generated by continuous stations owned by the Everglades Research Center. Hourly wind speed and direction measurements can only be generated by stations owned by any of the following agencies: Everglades Research Center, SFWMD, U.S. Army Corps of Engineers, NOAA.)

- **hour** — attribute of **HOURLY-HYDROLOGY-OBSERVATION**, range: 0..23 (key/3) (The hour during which the hydrology observation was made.)
- **date** — attribute of **HOURLY-HYDROLOGY-OBSERVATION**, range: Date (key/3) (The date on which the hourly hydrology observation was made.)

The objects of the category **HOURLY-HYDROLOGY-OBSERVATION** are identified by: hour date hourly-produced-by.

- **hourly-mean-stage** — attribute of **HOURLY-STAGE**, range: -99.99..+99.99 (m:1) (The hourly mean stage quantity measured in ft/100. That is, the value 1.23 means 0.0123 feet. This field is left blank when data is not available.)
- **hourly-precipitation-inch** — attribute of **HOURLY-RAINFALL**, range: 0.00..22.00 (m:1) (The hourly total precipitation quantity measured in inches. This field is left blank when data is not available.)
- **hourly-mean-wind-speed-mph** — attribute of **HOURLY-WIND**, range: 0.00..200.00 (m:1) (The mean wind speed for the hour, measured in miles per hour. This field is left blank when data is not available.)
- **hourly-wind-direction** — attribute of **HOURLY-WIND**, range: 1..360 (m:1) (The wind direction for the hour, measured in degrees. This field is left blank when data is not available.)

### 5.1.1.5. Monthly hydrology observations

Principal interviewer: Michael Alexopoulos. Client representative interviewed: David Sikema. Revised by Naphtali Rishe 03/19/91.

- **MONTHLY-HYDROLOGY-OBSERVATION** — category (A catalog of monthly hydrology observations which originate from stations within the Everglades National Park.)
- **MONTHLY-STAGE** — subcategory of **MONTHLY-HYDROLOGY-OBSERVATION** (Monthly mean stage measurements.)
- **MONTHLY-RAINFALL** — subcategory of **MONTHLY-HYDROLOGY-OBSERVATION** (Monthly total rainfall measurements.)
- **HYDROLOGY-STATION** — category (See subschema hydrology-stations.)
Every object of the category `MONTHLY-HYDROLOGY-OBSERVATION` must also belong to its subcategory. The following subcategories are disjoint: `MONTHLY-STAGE` and `MONTHLY-RAINFALL`.

- `monthly-produced-by` — relation from `MONTHLY-HYDROLOGY-OBSERVATION` to `HYDROLOGY-STATION` (key/3) (The station which generates monthly stage and/or rainfall measurements. Monthly stage and rainfall measurements can only be generated by discontinuous stations owned by the Everglades Research Center.)

- `month` — attribute of `MONTHLY-HYDROLOGY-OBSERVATION`, range: `1..12` (key/3) (The month during which a hydrology observation was made.)

- `year` — attribute of `MONTHLY-HYDROLOGY-OBSERVATION`, range: `1940..2040` (key/3) (The year during which a monthly hydrology observation was made.)

The objects of the category `MONTHLY-HYDROLOGY-OBSERVATION` are identified by: month year monthly-produced-by.
- **monthly-mean-stage** — attribute of **MONTHLY-STAGE**, range: -99.99..+99.99 (m:1) (The monthly mean stage quantity measured in ft/100. That is the value 1.23 means 0.0123 feet. This field is left blank when data is not available.)

- **monthly-precipitation-inch** — attribute of **MONTHLY-RAINFALL**, range: 0.00..22.00 (m:1) (The monthly total precipitation quantity measured in inches. This field is left blank when data is not available.)

### 5.1.1.6. Hydrology stations equipment

Principal interviewer: Michael Alexopoulos. Client representatives interviewed: De Witt Smith, David Sikema. Revised by Naphtali Rishe 03/19/91.

- **HYDROLOGY-STATION** — category (See subschema hydrology-stations.)

---

**Figure 5-7.** Semantic subschema for Hydrology Stations Equipment.

- **PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT** — category (A catalog of parameters for which data are collected from various equipment installed at different hydrology stations.)

Rishe-DDS p. 5.13
PARAMETER-DISCONTINUITY-PERIOD — category (A catalog of periods during which a parameter is not collected.)

EQUIPMENT — category (See subschema equipment.)

EQUIPMENT-INSTALLATION-PERIOD — category (A catalog of time intervals indicating the station where an equipment has been installed during that interval.)

collected-by — relation from PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT to EQUIPMENT (key/2) (The equipment which collects the parameter.)

the-parameter — relation from PARAMETER-DISCONTINUITY-PERIOD to PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT (key/2) (The parameter not collected for the period.)

installed-at — relation from EQUIPMENT-INSTALLATION-PERIOD to HYDROLOGY-STATION (key/3) (The station where the equipment was or is present during the time interval.)

the-equipment — relation from EQUIPMENT-INSTALLATION-PERIOD to EQUIPMENT (key/3) (The equipment installed at a station during a given time interval.)

begin-date — attribute of EQUIPMENT-INSTALLATION-PERIOD, range: Date (key/3) (The beginning date on which the equipment was installed at a specific station. May be omitted if unknown or irrelevant.)

end-date — attribute of EQUIPMENT-INSTALLATION-PERIOD, range: Date (m:1) (The ending date on which the equipment was removed from a specific station. Omitted when the equipment is still there.)

begin-date — attribute of PARAMETER-DISCONTINUITY-PERIOD, range: Date (key/2) (The beginning date on which data for a specific parameter ceased to be collected.)

end-date — attribute of PARAMETER-DISCONTINUITY-PERIOD, range: Date (m:1) (The last day of a period during which data for a specific parameter was not collected.)

reason-for-discontinuity — attribute of PARAMETER-DISCONTINUITY-PERIOD, range: String (m:1) (The reason for which data for a specific parameter was not collected during this time interval.)

parameter-name — attribute of PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT, range: Char(20) (key/2) (The name of the parameter collected by some specific equipment.)

parameter-description — attribute of PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT, range: String (m:1) (Designation of the parameter collected by some specific equipment.)

The objects of the category EQUIPMENT-INSTALLATION-PERIOD are identified by: begin-date installed-at the-equipment.

Rishe-DDS p. 5.14
The objects of the category **PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT** are identified by: parameter-name collected-by.

The objects of the category **PARAMETER-DISCONTINUITY-PERIOD** are identified by: begin-date the-parameter.

### 5.1.1.7. Fire history

Principal interviewer: Michael Alexopoulos. Client representatives interviewed: Sue Husari, Dave Lentz. Revised by Naphtali Rishe 03/19/91.

- **FIRE-INCIDENT** —category (A catalog of fire incidents which occur within the boundaries of the Everglades National Park.)
- **DAILY-DESCRIPTION** —category (Each fire occurring within the park may burn continuously for a number of days until it is put out. This category is a catalog of all the days for which a fire has lasted, for all fires.)
- **FIRE-WEATHER-OBSERVATION** —category (A catalog of fire-weather related observations for every day of interest to the Everglades Park authorities.)
- **HYDROLOGY-STATION** —category (See subschema hydrology.)
- **FIRE-WEATHER-STATION** —category (A catalog of fire-weather stations, where a fire-weather station is a collection of equipment used to measure various quantities pertaining to weather conditions within the Everglades Park area.)
- **YEAR** —category (A catalog of years during which various fire incidents occur.)
- **for** —relation from **DAILY-DESCRIPTION** to **FIRE-INCIDENT** (key/2) (The days during which a fire was burning.)
- **designated-hydrology-station** —relation from **FIRE-INCIDENT** to **HYDROLOGY-STATION** (m:1) (Closest hydrology station to the origin of the fire, designated by Park officials.)
- **observed-by** —relation from **FIRE-WEATHER-OBSERVATION** to **FIRE-WEATHER-STATION** (key/2) (The weather observations carried out by a fire-weather station every day.)
- **the-year** —relation from **FIRE-INCIDENT** to **YEAR** (key/2) (The year during which a fire incident has occurred.)
- **fire-number-within-year** —attribute of **FIRE-INCIDENT**, range: 1..999 (key/2) (A sequential integer value assigned to each fire incident within a calendar year. A fire incident occurs within a calendar year \( x \) if its starting date falls within \( x \).)
- **fire-type** —attribute of **FIRE-INCIDENT**, range: ‘incendiary’, ‘lightning’, ‘prescribed’, ‘research’ (m:1) (The possible type of a particular fire incident.)
- **fire-name** —attribute of **FIRE-INCIDENT**, range: String (m:1) (An arbitrary name assigned to each fire incident by the Everglades Park authorities.)
Figure 5-8. Semantic subschema for fire incident observations.
- **fire-origin-north** – attribute of FIRE-INCIDENT, range: 2746840..2865840 (m:1) (UTM coordinate for North. Together with fire-origin-east they indicate the geographical position of the origin of a fire incident. *May be overwritten by GIS.*)

- **fire-origin-east** – attribute of FIRE-INCIDENT, range: 446880..563280 (m:1) (UTM coordinate for East. Together with fire-origin-north they indicate the geographical position of the origin of a fire incident. *May be overwritten by GIS.*)

- **fuel-model** – attribute of FIRE-INCIDENT, range: ‘n’, ‘d’ (m:1) (Dominant vegetation where fire incident occurred. n stands for Pine, and d for Grass.)

- **fine-fuel-load** – attribute of FIRE-INCIDENT, range: 0..9999 (m:1) (The amount of fuel per square meter for a fire incident, measured in grams per square meter.)

- **cost** – attribute of FIRE-INCIDENT, range: 0..99999999 (m:1) (Total cost in dollars to manage fire. Rounded to nearest dollar.)

- **year-number** – attribute of YEAR, range: 1940..2040 (key) (The year number.)

- **is-drought-year** – attribute of YEAR, range: Boolean (m:1)

- **has-crossed-perimeter** – attribute of FIRE-INCIDENT, range: Boolean (m:1) (Descriptor of geographical zones in the park or area surrounding it.)

- **date** – attribute of DAILY-DESCRIPTION, range: Date (key/2) (The date of a particular day during which a fire was still burning. The year in this date should be the same or one more than the year during which the fire incident occurred.)

- **new-acres-burned** – attribute of DAILY-DESCRIPTION, range: 0..99999 (m:1) (The number of acres burned only for that particular date indicated in the Date attribute.)

- **soil-moisture** – attribute of DAILY-DESCRIPTION, range: 0..100 (m:1) (Ratio (wet in grams)−(dry in grams))/(dry in grams)*100%. Quantity indicating the soil moisture during a fire day. Percentage measure.)

- **date** – attribute of FIRE-WEATHER-OBSERVATION, range: Date (key/2) (The date of the day for which the weather observation has been made.)

- **state-of-the-weather** – attribute of FIRE-WEATHER-OBSERVATION, range: 0..9 (m:1) (An arbitrarily assigned value for the state of the weather; 0 means clear, 9 means thunder, etc. Assigned by the Everglades Park personnel.)

- **dry-temperature** – attribute of FIRE-WEATHER-OBSERVATION, range: 0..110 (m:1) (Measured in degrees Fahrenheit.)

- **relative-humidity** – attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Percentage measure of relative humidity.)

- **relative-humidity-max** — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Maximum value of relative humidity for the day. Percentage measure.)
relative-humidity-min — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Minimum value of relative humidity for the day. Percentage measure.)

wind-direction — attribute of FIRE-WEATHER-OBSERVATION, range: 0..8 (m:1) (0-8 compass point.)

wind-speed-mph — attribute of FIRE-WEATHER-OBSERVATION, range: 0..99 (m:1) (Measured in miles per hour.)

temperature-max — attribute of FIRE-WEATHER-OBSERVATION, range: 20..110 (m:1) (Maximum temperature for the day. Measured in degrees Fahrenheit.)

temperature-min — attribute of FIRE-WEATHER-OBSERVATION, range: 20..110 (m:1) (Minimum temperature for the day. Measured in degrees Fahrenheit.)

precipitation-duration — attribute of FIRE-WEATHER-OBSERVATION, range: 0..24 (m:1) (Precipitation duration measured in hours.)

precipitation-amount-inch — attribute of FIRE-WEATHER-OBSERVATION, range: 0..99.99 (m:1) (Precipitation amount measured in inches.)

drought-index — attribute of FIRE-WEATHER-OBSERVATION, range: 0..999 (m:1) (The drought index for the day.)

live-fuel-moisture — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Live fuel moisture. Percentage measure.)

thousand-hour-fuel-moisture-n — attribute of FIRE-WEATHER-OBSERVATION, range: 0..99 (m:1) (One thousand hour fuel moisture for Pine. Percentage measure.)

thousand-hour-fuel-moisture-d — attribute of FIRE-WEATHER-OBSERVATION, range: 0..99 (m:1) (One thousand hour fuel moisture for Grass. Percentage measure.)

ignition-component-n — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Fire ignition component for Pine.)

ignition-component-d — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Fire ignition component for Grass.)

spread-component-n — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Fire spread component for Pine.)

spread-component-d — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Fire spread component for Grass.)

energy-release-component-n — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Energy release component for Pine.)

energy-release-component-d — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Energy release component for Grass.)
burning-index-n — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Burning index for Pine.)

burning-index-d — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Burning index for Grass.)

fire-load-index-n — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Fire load index for Pine.)

fire-load-index-d — attribute of FIRE-WEATHER-OBSERVATION, range: 0..100 (m:1) (Fire load index for Grass.)

The objects of the category FIRE-INCIDENT are identified by: fire-number-within-year the-year.

The objects of the category DAILY-DESCRIPTION are identified by: date for.

The objects of the category YEAR are identified by: year-number.

The objects of the category FIRE-WEATHER-STATION are identified by: station-id.

The objects of the category FIRE-WEATHER-OBSERVATION are identified by: date observed-by.

station-id — attribute of FIRE-WEATHER-STATION, range: Char(8) (key) (A sequence of alphanumeric characters which is assigned to each fire-weather station and which uniquely identifies that station.)

station-description — attribute of FIRE-WEATHER-STATION, range: String (m:1) (English name or designation of the station.)

location-north — attribute of FIRE-WEATHER-STATION, range: 2746840..2865840 (m:1) (UTM north coordinate of a fire-weather station.)

location-east — attribute of FIRE-WEATHER-STATION, range: 446880..563280 (m:1) (UTM east coordinate of a fire-weather station.)
### 5.1.2. Relational schema of the application

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BATTERY-CHANGE</strong></td>
<td>in-equipment--park-service-number-in-key:Char(15); battery-change-date-in-key:Date;</td>
</tr>
</tbody>
</table>

Reference from **BATTERY-CHANGE** to **EQUIPMENT**: in-equipment--park-service-number-in-key→park-service-number-key.

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CALIBRATION</strong></td>
<td>equipment-calibrated--park-service-number-in-key:Char(15); calibration-date-in-key:Date;</td>
</tr>
<tr>
<td></td>
<td>calibration-method:'Field’, ‘Lab’; is-calibration-using-other-technique:Boolean; technique-description:String;</td>
</tr>
</tbody>
</table>

Reference from **CALIBRATION** to **EQUIPMENT**: equipment-calibrated--park-service-number-in-key→park-service-number-key.

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DAILY-DESCRIPTION</strong></td>
<td>for--year-number-in-key:1940..2040; for--fire-number-within-year-in-key:1..999; date-in-key:Date; new-acres-burned:0..99999; soil-moisture:0..100;</td>
</tr>
</tbody>
</table>

Reference from **DAILY-DESCRIPTION** to **FIRE-INCIDENT**: for--year-number-in-key→year-number-in-key, for--fire-number-within-year-in-key→fire-number-within-year-in-key.

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EQUIPMENT</strong></td>
<td>park-service-number-key:Char(15) 1:1; model-number:Char(15); serial-number:Char(20); type-descriptor:Char(30);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EQUIPMENT-INSTALLATION-PERIOD</strong></td>
<td>equipment--park-service-number-in-key:Char(15); installed-at--station-id-in-key:Char(15); begin-date-in-key:Date; end-date:Date;</td>
</tr>
</tbody>
</table>

Reference from **EQUIPMENT-INSTALLATION-PERIOD** to **HYDROLOGY-STATION**: installed-at--station-id-in-key→station-id-key.

Reference from **EQUIPMENT-INSTALLATION-PERIOD** to **EQUIPMENT**: equipment-park-service-number-in-key→park-service-number-key.
## FIRE-INCIDENT

- `year-number-in-key`: 1940..2040
- `fire-number-within-year-in-key`: 1..999
- `fine-fuel-load`: 0..9999
- `fire-management-unit`: 1..44
- `fire-name`: String
- `cost`: 0..99999999
- `fire-origin-east`: 446880..563280
- `fire-origin-north`: 2746840..2865840
- `fire-type`: ‘incendiary’, ‘lightning’, ‘prescribed’, ‘research’
- `fuel-model`: ‘n’, ‘d’
- `has-crossed-perimeter`: Boolean
- `designated-hydrology--station-id`: Char(15)

Reference from `FIRE-INCIDENT` to `HYDROLOGY-STATION`: designated-hydrology--station-id → station-id-key.

Reference from `FIRE-INCIDENT` to `YEAR`: year-number-in-key → year-number-key.

## FIRE-WEATHER-OBSERVATION

- `observed-by--station-id-in-key`: Char(8)
- `date-in-key`: Date
- `burning-index-n`: 0..100
- `drought-index`: 0..999
- `dry-temperature`: 0..110
- `energy-release-component-d`: 0..100
- `energy-release-component-n`: 0..100
- `fire-load-index-d`: 0..100
- `fire-load-index-n`: 0..100
- `ignition-component-d`: 0..100
- `ignition-component-n`: 0..100
- `live-fuel-moisture`: 0..100
- `precipitation-amount-inch`: 0.0..99.99
- `precipitation-duration`: 0..24
- `relative-humidity`: 0..100
- `relative-humidity-max`: 0..100
- `relative-humidity-min`: 0..100
- `spread-component-d`: 0..100
- `spread-component-n`: 0..100
- `state-of-weather`: 0..9
- `temperature-max`: 20..110
- `temperature-min`: 20..110
- `thousand-hour-fuel-moisture-d`: 0..99
- `thousand-hour-fuel-moisture-n`: 0..99
- `wind-direction`: 0..8
- `wind-speed-mph`: 0..99

Reference from `FIRE-WEATHER-OBSERVATION` to `FIRE-WEATHER-STATION`: observed-by--station-id-in-key → station-id-key.

## FIRE-WEATHER-STATION

- `station-id-key`: Char(8)
- `location-north`: 2746840..2865840
- `station-description`: String
- `location-east`: 446880..563280

## HYDROLOGY-STATION

- `station-id-key`: Char(15)
- `station-description`: String
- `location-tolerance-ft`: 0..1000
- `station-location-east`: 446880..563280
- `station-location-north`: 2746840..2865840
- `is-continuous-station`: Boolean
- `is-discontinuous-station`: Boolean
- `benchmark-location-east`: 446880..563280
- `benchmark-location-north`: 2746840..2865840

Rishe-DDS p. 5.21
PARAMETER-DISCONTINUITY-PERIOD

| parameter--collected-by--park-service-number-in-key:Char(15); |
| parameter-name-in-key:Char(20); | begin-date-in-key:Date; end-date:Date; |
| reason-for-discontinuity:String; |

Reference from PARAMETER-DISCONTINUITY-PERIOD to PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT: parameter--collected-by--park-service-number-in-key→collected-by--park-service-number-in-key, parameter-name-in-key→parameter-name-in-key.

PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT

| collected-by--park-service-number-in-key:Char(15); |
| parameter-name-in-key:Char(20); |
| parameter-description:String; |

Reference from PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT to EQUIPMENT: collected-by--park-service-number-in-key→park-service-number-key.

REPAIR-PERIOD

| equipment-repaired--park-service-number-in-key:Char(15); |
| begin-date-in-key:Date; end-date:Date; reason-for-repair:String; repair-entity:String; |

Reference from REPAIR-PERIOD to EQUIPMENT: equipment-repaired--park-service-number-in-key→park-service-number-key.

STATION-CONTINUITY-PERIOD

| discontinuous--station-id-in-key:Char(15); begin-date-in-key:Date; end-date:Date; |

Reference from STATION-CONTINUITY-PERIOD to HYDROLOGY-STATION: discontinuous--station-id-in-key→station-id-key.
YEAR
year-number-key: 1940..2040 1:1; is-drought-year: Boolean;

DAILY-DISCHARGE
daily-produced-by--station-id-in-key: Char(15); date-in-key: Date;
daily-downstream-stage-ft: 0..12; daily-mean-discharge: -99999.99..+99999.99;
daily-upstream-stage-ft: 0..12; formula: ‘weir’, ‘rating-curves’;

Reference from DAILY-DISCHARGE to HYDROLOGY-STATION: daily-produced-by--station-id-in-key→station-id-key.

DAILY-EVAPORATION
daily-produced-by--station-id-in-key: Char(15); date-in-key: Date;
total-daily-evaporation-inch: 0.00..12.00;

Reference from DAILY-EVAPORATION to HYDROLOGY-STATION: daily-produced-by--station-id-in-key→station-id-key.

DAILY-RAINFALL
daily-produced-by--station-id-in-key: Char(15); date-in-key: Date;
daily-precipitation-inch: 0.00..22.00;

Reference from DAILY-RAINFALL to HYDROLOGY-STATION: daily-produced-by--station-id-in-key→station-id-key.

DAILY-STAGE
daily-produced-by--station-id-in-key: Char(15); date-in-key: Date;
daily-mean-stage: 99.99..+99.99;

Reference from DAILY-STAGE to HYDROLOGY-STATION: daily-produced-by--station-id-in-key→station-id-key.
### DAILY-TEMPERATURE

- `daily-produced-by--station-id-in-key`: Char(15);
- `date-in-key`: Date;
- `daily-max-temperature`: 20.00..120.00;
- `daily-mean-temperature`: 20.00..120.00;
- `daily-min-temperature`: 20.00..120.00;

Reference from **DAILY-TEMPERATURE** to **HYDROLOGY-STATION**: daily-produced-by--station-id-in-key→station-id-key.

### HOURLY-RAINFALL

- `hourly-produced-by--station-id-in-key`: Char(15);
- `hour-in-key`: 0..23;
- `date-in-key`: Date;
- `hourly-precipitation-inch`: 0.00..22.00;

Reference from **HOURLY-RAINFALL** to **HYDROLOGY-STATION**: hourly-produced-by--station-id-in-key→station-id-key.

### HOURLY-STAGE

- `hourly-produced-by--station-id-in-key`: Char(15);
- `hour-in-key`: 0..23;
- `date-in-key`: Date;
- `hourly-mean-stage`: -99.99..+99.99;

Reference from **HOURLY-STAGE** to **HYDROLOGY-STATION**: hourly-produced-by--station-id-in-key→station-id-key.

### HOURLY-WIND

- `hourly-produced-by--station-id-in-key`: Char(15);
- `hour-in-key`: 0..23;
- `date-in-key`: Date;
- `hourly-mean-wind-speed-mph`: 0.00..200.00;
- `hourly-wind-direction`: 1..360;

Reference from **HOURLY-WIND** to **HYDROLOGY-STATION**: hourly-produced-by--station-id-in-key→station-id-key.

### MONTHLY-RAINFALL

- `monthly-produced-by--station-id-in-key`: Char(15);
- `month-in-key`: 1..12;
- `year-in-key`: 1940..2040;
- `monthly-precipitation-inch`: 0.00..22.00;

Reference from **MONTHLY-RAINFALL** to **HYDROLOGY-STATION**: monthly-produced-by--station-id-in-key→station-id-key.

Rishe-DDS p. 5.24
### MONTHLY-STAGE

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>monthly-produced-by--station-id-in-key:Char(15);</td>
<td>month-in-key:1..12;</td>
</tr>
<tr>
<td>year-in-key:1940..2040; monthly-mean-stage:-99.99..+99.99;</td>
<td></td>
</tr>
</tbody>
</table>
```

Reference from **MONTHLY-STAGE** to **HYDROLOGY-STATION**: monthly-produced-by--station-id-in-key → station-id-key.

### CALIBRATION--HAS-USED--EQUIPMENT

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>calibration-equipment-calibrated--park-service-number-in-key:Char(15);</td>
<td></td>
</tr>
<tr>
<td>calibration-date-in-key:Date; equipment--park-service-number-in-key:Char(15);</td>
<td></td>
</tr>
</tbody>
</table>
```

Reference from **CALIBRATION HAS-USED EQUIPMENT** to **CALIBRATION**: calibration-equipment-calibrated--park-service-number-in-key → equipment-calibrated--park-service-number-in-key, calibration-date-in-key → calibration-date-in-key.

Reference from **CALIBRATION HAS-USED EQUIPMENT** to **EQUIPMENT**: equipment--park-service-number-in-key → park-service-number-key.

Some of the Integrity Constraints Generated During Schema Conversion

(For every `x` in **CALIBRATION**: if not `x` technique-description null then `x`.is-calibration-using-other-technique) and

(For every `x` in **HYDROLOGY-STATION**: if not `x` begin-date null then `x`.is-temporary-station) and

(For every `x` in **HYDROLOGY-STATION**: if not `x` end-date null then `x`.is-temporary-station) and

(For every `x` in **CALIBRATION--HAS-USED--EQUIPMENT**: exists `y` in **CALIBRATION**: `x`.the-calibration-equipment-calibrated--park-service-number-in-key = `y`.the-equipment-calibrated--park-service-number-in-key and `x`.the--calibration-date-in-key = `y`.calibration-date-in-key) and

(For every `x` in **CALIBRATION--HAS-USED--EQUIPMENT**: exists `y` in **EQUIPMENT**: `x`.the-equipment--park-service-number-in-key = `y`.park-service-number-key) and

(For every `x` in **PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT**: exists `y` in **EQUIPMENT**: `x`.collected-by--park-service-number-in-key = `y`.park-service-number-key) and

(For every `x` in **FIRE-INCIDENT**: exists `y` in **HYDROLOGY-STATION**: `x` designated-hydrology--station-id null or `x`.designated-hydrology--station-id = `y`.station-id-key) and

Rishe-DDS p. 5.25
(for every x in DAILY-DESCRIPTION: exists y in FIRE-INCIDENT: (x.for--year-number-in-key = y.the--year-number-in-key and x.for--fire-number-within-year-in-key = y.fire-number-within-year-in-key)) and

(for every x in BATTERY-CHANGE: exists y in EQUIPMENT: x.in-equipment--park-service-number-in-key = y.park-service-number-key) and

(for every x in EQUIPMENT-INSTALLATION-PERIOD: exists y in HYDROLOGY-STATION: x.installed-at--station-id-in-key = y.station-id-key) and

(for every x in FIRE-WEATHER-OBSERVATION: exists y in FIRE-WEATHER-STATION: x.observed-by--station-id-in-key = y.station-id-key) and

(for every x in STATION-CONTINUITY-PERIOD: exists y in HYDROLOGY-STATION: x.the-discontinuous--station-id-in-key = y.station-id-key and y.is-fixed-station) and

(for every x in EQUIPMENT-INSTALLATION-PERIOD: exists y in EQUIPMENT: x.the-equipment--park-service-number-in-key = y.park-service-number-key) and

(for every x in CALIBRATION: exists y in EQUIPMENT: x.the-equipment-calibrated--park-service-number-in-key = y.park-service-number-key) and

(for every x in REPAIR-PERIOD: exists y in EQUIPMENT: x.the-equipment-repaired--park-service-number-in-key = y.park-service-number-key) and

(for every x in PARAMETER-DISCONTINUITY-PERIOD: exists y in PARAMETER-MEASURED-BY-SPECIFIC-EQUIPMENT: (x.the-parameter--collected-by--park-service-number-in-key = y.collected-by--park-service-number-in-key and x.the--parameter-name-in-key = y.parameter-name-in-key)) and

(for every x in FIRE-INCIDENT: exists y in YEAR: x.the--year-number-in-key = y.year-number-key) and

(for every x in DAILY-DISCHARGE: exists y in HYDROLOGY-STATION: x.daily-produced-by--station-id-in-key = y.station-id-key) and

(for every x in DAILY-EVAPORATION: exists y in HYDROLOGY-STATION: x.daily-produced-by--station-id-in-key = y.station-id-key) and

(for every x in DAILY-RAINFALL: exists y in HYDROLOGY-STATION: x.daily-produced-by--station-id-in-key = y.station-id-key) and

(for every x in DAILY-STAGE: exists y in HYDROLOGY-STATION: x.daily-produced-by--station-id-in-key = y.station-id-key) and

(for every x in DAILY-TEMPERATURE: exists y in HYDROLOGY-STATION: x.daily-produced-by--station-id-in-key = y.station-id-key) and

Rishe-DDS p. 5.26
(for every \( x \) in \( \text{HOURLY-RAINFALL} \): exists \( y \) in \( \text{HYDROLOGY-STATION} \): \( x.\text{hourly-produced-by--station-id-in-key} = y.\text{station-id-key} \)) and

(for every \( x \) in \( \text{HOURLY-STAGE} \): exists \( y \) in \( \text{HYDROLOGY-STATION} \): \( x.\text{hourly-produced-by--station-id-in-key} = y.\text{station-id-key} \)) and

(for every \( x \) in \( \text{HOURLY-WIND} \): exists \( y \) in \( \text{HYDROLOGY-STATION} \): \( x.\text{hourly-produced-by--station-id-in-key} = y.\text{station-id-key} \)) and

(for every \( x \) in \( \text{MONTHLY-RAINFALL} \): exists \( y \) in \( \text{HYDROLOGY-STATION} \): \( x.\text{monthly-produced-by--station-id-in-key} = y.\text{station-id-key} \)) and

(for every \( x \) in \( \text{MONTHLY-STAGE} \): exists \( y \) in \( \text{HYDROLOGY-STATION} \): \( x.\text{monthly-produced-by--station-id-in-key} = y.\text{station-id-key} \))

5.2. Flow of Database Design

Figure 5-9 is an information flow diagram that outlines the major steps of database application design, including the schema, integrity constraints, userviews, data manipulation programs, query forms, and ad hoc queries. The design proceeds in the direction of the arrows from semantic descriptions to descriptions in the conventions and languages supported by the available DBMS. Nodes marked in brackets are omitted for some DBMSs.

In the first step, a conceptual schema of an enterprise is designed using the Semantic Binary Model. Then the schema is converted into the relational, network, or hierarchical model by manual algorithms. According to the criteria described in the text to assess the quality of databases, these manual algorithms produce very high-quality results.

5.3. Other Methodologies

5.3.1. An alternative methodology: normalization

Normalization is a methodology for the design of relational databases. This methodology was used to be quite popular in the academic world. However, it has rarely been used in the application industry. One of the reasons for its lack of popularity in the industry is the mathematical sophistication of the normalization methodology.

This is a “bottom-up” methodology. The design proceeds as follows. First, a poor relational schema is designed directly from the requirements. Then, the schema is refined in steps by eliminating certain aspects of redundancy (and thus potential inconsistency and update anomalies). At every step the schema satisfies certain mathematically defined criteria of nonredundancy corresponding to that step. These criteria are called normal forms.

1. The initial schema is said to be in the first normal form.
2. The product of the first step, satisfying certain broad criteria, is in the second normal form.
3. The product of the next step, satisfying certain stricter criteria, is in the third normal form.

Risque-DDS p. 5.27
4. The normalization process can continue further until the arsenal of normal form definitions is exhausted.

After the design is completed, all the schemas but the last one are discarded. Programs, queries, and the like are designed directly in terms of the final schema. Figure 5-10 is a

Rishe-DDS p. 5.28
diagram of the flow of design by normalization.

Figure 5-10. Alternative relational database design: Normalization.

5.3.1.1. *The third normal form defined

The following is a definition of the third normal form.

An attribute $A$ of table $T$ is said to be **functionally dependent** on a set of attributes \( \{B_1, \ldots, B_k\} \) of $T$ if for no tuple of values \( (b_1, \ldots, b_k) \) of these attributes there may be two different values of $A$ in the table at the same time.

A table $T$ having exactly one key (possibly a multiattribute key) is said to be in the **third normal form** if:

No nonkey attribute $A$ is functionally dependent on any set of attributes, unless the latter set of attributes contains $A$ or contains the whole key.

(The definition is more complex for the unlikely case that the table has more than one key.)

For example, let $T$ be a table with four attributes $A$, $B$, $C$, and $D$. Let the key of $T$ be \( \{A, B\} \). If $T$ is in the third normal form then:

- $a$. $A$ is not functionally dependent on \( \{B, C, D\} \)
- $b$. $B$ is not functionally dependent on \( \{A, C, D\} \)
c. \( C \) is not functionally dependent on \( \{ A, D \} \)

d. \( C \) is not functionally dependent on \( \{ B, D \} \)

e. \( D \) is not functionally dependent on \( \{ A, C \} \)

f. \( D \) is not functionally dependent on \( \{ B, C \} \)

If any of these conditions were violated, there would be a clear redundancy. For example, if \( D \) is functionally dependent on \( \{ B, C \} \), then observe the redundancy in the following instantaneous table. We can deduce from the constraint that the \( ? \) in the second row should read 45:

\[
\begin{array}{cccc}
A & B & C & D \\
37 & 15 & 5 & 45 \\
12 & 15 & 5 & ? \\
\end{array}
\]

5.3.2. A comparison of methodologies

1. The normalization methodology captures only a few of the aspects of the semantic quality of databases, while the methodology suggested in this text attempts to capture all of the aspects.

2. The normalization methodology is too difficult to be used by most systems analysts and software engineers.

3. The normalization methodology is bottom-up: a "bad" database is designed, and then it is refined by normalization. This is analogous to writing a bad program and then improving its structure.

   This book’s methodology is top-down: good semantic schemas are designed first and then they are downgraded to meet implementational restrictions, while the original semantic schemas remain to serve as documentation. This is analogous to writing an algorithm first and then translating it into a structured program, while the algorithm remains as documentation.

   Figure ____ (page ____ ) is a “schema” of the world of relational database schemas and two database design processes: the binary-relational conversion according to the methodology of this book and the schema normalization according to the alternative methodology. This “methodology schema” shows:

   • Every relational schema is a binary schema.
   • Some of the binary schemas, which are not relational schemas, are high-quality schemas according to all the criteria of schema quality.
   • Some of the relational schemas satisfy to a certain degree some of the nonredundancy criteria. These limited criteria are primarily concerned with the possible redundancy of a table which could be split into two tables. The relational schemas which satisfy these limited criteria to a certain minimal degree (at least) are called the second-normal-form schemas. Some of the second-normal-form schemas satisfy the limited criteria to a higher degree and correspondingly belong to the THIRD-NORMAL-FORM, FOURTH-
Every high-quality relational schema should be in the maximal known normal form, but the opposite is not true: some maximal-normal-form schemas are not of high quality, since they do not address all the quality criteria.

The process of normalization begins with a very poor relational schema and converts it into a second-normal-form schema, then into a third normal form schema, and so on.

The methodology of this book begins with a quality semantic schema and downgrades it to a quality relational schema.

Figure ___ outlines the “instantaneous database” under the “methodology schema.” The individual schemas are points there in a coordinate system of schema quality.
This diagram depicts the "instantaneous database" for the "schema" of conversion methodologies. The database schemas $s_1$, $s_2$, $s_3$, and $s_4$ are individual objects in that "instantaneous database."

**Figure 5-11.** A comparison between database design methodologies.
Figure 5-12. A “schema” of conversion methodologies: normalization versus semantic-to-relational conversion. The arrows with dashes show the subcategories among the categories of schemas.
Reference Schemas

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.

![Diagram of semantic schema for university application]

**Figure Ref-1.** A semantic schema for a university application.

Rishe-DDS p. 5.34
Figure Ref-2. A relational schema for the university application.
6. FROM THE SEMANTIC TO THE NETWORK MODEL

This chapter defines the Network Data Model and adapts the top-down database design methodology to network databases. Section 3 of this chapter discusses network database languages: application of the generic fourth-generation and logic-based languages and a special navigational language for the Network Model.

Since the 1970s, the Network Data Model has been very popular in the industry. An alternative name of the model is CODASYL/DBTG, after the name of the committee that produced a standard for the Network Model (CODASYL Data Base Task Group).

6.1. Definitions

Orderless network schema — a binary schema satisfying the following:

a. All the abstract categories are pairwise disjoint.

b. Every relation is either an attribute (an m:1 relation to a concrete category), or a 1:m relation between different abstract categories.
Figure 6-1. An orderless network schema for the university application.
Example 6-1.
The schema on Figure ___ could be an orderless network schema for a university, provided there are no persons but students and instructors, and the categories INSTRUCTOR and STUDENT are disjoint.

Default names of relations

It is customary in the network model to name relations as
domain hyphen range

Of course, this convention may be used only when there is no other relation between the same domain and range.

In the graphic representation of network schemas we may omit some of the names of the relations. The omitted names by default conform to the above convention.

Example 6-2.

□ instructor-work — relation from INSTRUCTOR to WORK (1:m)

Onto relation—a relation whose inverse is total.

This means that a relation $R$ from $C_1$ to $C_2$ is onto if for every object $y$ of $C_2$ there is an object $x$ of $C_1$ such that $xRy$.

Example 6-3.

□ — relation from STUDENT to ENROLLMENT (1:m,onto)
(It is onto because every enrollment has a student related to it.)
□ — relation from INSTRUCTOR to WORK (1:m,onto)
(It is onto because every event of work is related to an instructor.)

The phrase ‘‘relation $R$ is onto category $C$’’ means:

$R$ is an onto relation, and its range is $C$.

The phrase ‘‘category $C_2$ depends on category $C_1$’’ means:

there is a relation from $C_1$ onto $C_2$

(that is, there is a relation which relates every object of $C_2$ to an object of $C_1$).

Rishe-DDS p. 6.3
Figure 6-2. $C_2$ depends on $C_1$.

Example 6-4.

- OFFERING depends on COURSE
- OFFERING depends on QUARTER
- OFFERING depends on INSTRUCTOR
- ENROLLMENT depends on OFFERING
- ENROLLMENT depends on STUDENT
- DEPARTMENT-NAMING depends on DEPARTMENT
- WORK depends on DEPARTMENT
- WORK depends on INSTRUCTOR

The phrase “category $C_1$ indirectly depends on category $C_2$” means (recursively):

$C_1$ depends on $C_2$, or there exists $C_3$ such that $C_1$ depends on $C_3$ and $C_3$ indirectly depends on $C_2$.

Example 6-5.

- ENROLLMENT indirectly depends on OFFERING
  (since ENROLLMENT depends on OFFERING).
- ENROLLMENT indirectly depends on COURSE

Independent category—an abstract category which depends on no category.

Example 6-6.

Independent categories:

STUDENT, INSTRUCTOR, DEPARTMENT, COURSE, QUARTER.

Ordered network schema—a binary system consisting of:
a. An orderless network schema.

b. A category, called SYSTEM, in which at all times there is only one and the same object SYSTEM.

c. Relations from SYSTEM onto some of the abstract categories of the orderless schema, such that every abstract category of the orderless schema would be indirectly dependent on SYSTEM.

Example 6-7.

- relation from SYSTEM to STUDENT (1:m, onto)
- relation from SYSTEM to INSTRUCTOR (1:m, onto)
- relation from SYSTEM to DEPARTMENT (1:m, onto)
- relation from SYSTEM to COURSE (1:m, onto)
- relation from SYSTEM to QUARTER (1:m, onto)

d. For every relation $R$ between different abstract categories $C_1$ and $C_2$ there is a relation $NEXT_R$ such that:

- The domain of $NEXT_R$ is $C_2$.
- The range of $NEXT_R$ is $C_2$.
- For every object $x$ in $C_1$, $NEXT_R$ constitutes a linear order of $C_2$’s objects connected by $R$ to $x$.

Example 6-8.

next-system-department → relation from DEPARTMENT to DEPARTMENT (1:1)

This is a chain connecting all of the departments:

$d_1 \rightarrow d_2 \rightarrow d_3 \rightarrow d_4$

Example 6-9.

next-system-student → relation from STUDENT to STUDENT (1:1)
This is a chain connecting all of the students:

\[ s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \rightarrow s_{500} \]

Example 6-10.

- next-major-st – relation from STUDENT to STUDENT (1:1)

This is a set of chains, one chain per department, connecting all the majoring students of the department.

For department \( d_1 \): 

\[ s_{11} \rightarrow s_{25} \rightarrow s_3 \rightarrow s_{15} \]

For department \( d_2 \): trivial chain because the department has only 1 majoring student.

For department \( d_3 \): empty chain because the department has no majoring students.

For department \( d_4 \): 

\[ s_1 \rightarrow s_{220} \rightarrow s_{31} \]

(The remaining 492 students have not declared their majors.)

Network schema terminology

The following terms are frequently used in network database management systems. Most of them have synonyms in the binary terminology.

**Record-type** — abstract category.

*Example 6-11.*

DEPARTMENT is a record-type.

**Field** – attribute.

*Example 6-12.*

LAST-NAME is a field.

**Set-type** – a nonattribute relation between different categories.

*Example 6-13.*

- major-st – relation from DEPARTMENT to STUDENT (1:m)
- instructor-work – relation from INSTRUCTOR to WORK (1:m)
**Record occurrence** — a part of an instantaneous database, consisting of exactly one abstract object and all its attributes.

*Example 6-14.*
A record occurrence of record-type *STUDENT*:

```
ONE student
  last-name: Jackson
  first-name: Mary
  birth-year: 1970
  address: 123 Dorms
```

**Set occurrence** of a set-type $R$ for an object $x$ — a part of an instantaneous database consisting only of $x$, all the objects related to $x$ by $R$, and the relation $NEXT_R$ on these objects.

*Example 6-15.*
Let $s_1$, $s_2$, $s_3$, $s_4$ be the only majoring students of department $d$. Then the following may be a set occurrence:

$$d, s_1 NEXT_{\text{major-st}} s_2 NEXT_{\text{major-st}} s_3 NEXT_{\text{major-st}} s_4$$

Each set-type whose domain is *SYSTEM* has exactly one set-occurrence, since there is only one object of the record-type *SYSTEM*.

*Example 6-16.*
The following is the set-occurrence of *SYSTEM-STUDENT* in an instantaneous database.
Example 6-17.
The following figure shows a part of a network instantaneous database. This part contains all the record-occurrences of *STUDENT*, *DEPARTMENT*, and *DEPARTMENT-NAMING* and all the set-occurrences of *SYSTEM-STUDENT*, *SYSTEM-DEPARTMENT*, *DEPARTMENT--DEPARTMENT-NAMING*, *MAJOR-ST*, and *MINOR-ST*.
Owner of a set-type $R$ — its domain.
Example 6-18.
DEPARTMENT is the owner of MAJOR-ST.

Member of a set-type $R$ — its range.

Example 6-19.
STUDENT is the member of MAJOR-ST.

Order-type of a set type $R$ — the integrity constraints, the implementational restrictions, or the inference rules regarding the ordering of a set-occurrence (as determined by the relation $NEXT_R$).

Many network database management systems recognize the following order types:

‘order is last’ — the member object last related (in time) to a given owner object becomes the last in the order among all the member objects related by the relation to the owner object.

Example 6-20.
Let the following be the set occurrence of MAJOR-ST for department $d$:

$$d, s_1 \ NEXT_{\text{major-st}} s_2 \ NEXT_{\text{major-st}} s_3 \ NEXT_{\text{major-st}} s_4$$

Now assume that another student $s$ becomes a majoring student of $d$. Then the new set occurrence would be

$$d, s_1 \ NEXT_{\text{major-st}} s_2 \ NEXT_{\text{major-st}} s_3 \ NEXT_{\text{major-st}} s_4 \ NEXT_{\text{major-st}} s$$

‘order is first’ — the member object last (in time) related to a given owner object becomes the first in the order among all the member objects related by the relation to the owner object.

Example 6-21.
Let the following be the set occurrence of MAJOR-ST for department $d$:

$$d, s_1 \ NEXT_{\text{major-st}} s_2 \ NEXT_{\text{major-st}} s_3 \ NEXT_{\text{major-st}} s_4$$

Now assume that another student $s$ becomes a majoring student of $d$. Then the new set occurrence would be
`order is ascending by field f` — the member object last related to a given owner object is put in a position within the order according to the following criterion for every two member objects \( y_1 \) and \( y_2 \):

\[
\text{if } y_1 \text{ NEXT}_R y_2 \text{ then } y_1.f \leq y_2.f
\]

This order type is often also called **order by key**. This “key” has nothing to do with the concept of *key* as defined for the relational model. To avoid confusion, the "order by key" terminology is not used in this text.

`order is descending by field f` — the member object last related to a given owner object is put in a position within the order according to the following criterion for every two member objects \( y_1 \) and \( y_2 \):

\[
\text{if } y_1 \text{ NEXT}_R y_2 \text{ then } y_1.f \geq y_2.f
\]

`order is \( o_1, o_2, \ldots, o_k \)`’, where each \( o_i \) is of the form

- `ascending by field \( f_i \)` or `descending by field \( f_i \)`

— a lexicographic combination of ordering conditions on several fields.

`order is current` — no constraints, restrictions, or program-independent rules for the order. Pragmatically, the position of a new object is determined by the application program: the position will be right next to the last object accessed by the program within the same set-occurrence.

### 6.2. Database Design

**Conversion algorithm** of a semantic schema into a network schema whose quality is among the highest possible for the latter (provided the original semantic schema is of high quality):

1. **Convert all abstract categories into disjoint ones** as in the conversion for the Relational Model.

   **Note:**

   - In the Network Model, the categories need not have keys. Thus, when the solution **Events** is chosen, its redundancy cannot be controlled in terms of the key. If the supercategory has a 1:1 attribute, then we can specify an integrity constraint controlling the **Events** redundancy in terms of that attribute.

   **Example 6-22.**

   Suppose we had

   - \( ssn \) — attribute of **PERSON**, range: *Integer* (1:1)

   After the conversion into **Events** we would have
In this case, we can specify a constraint for every relation whose domain or range was the category PERSON. For the relation ADDRESS such a constraint is:

```
for every s in STUDENT:
for every i in INSTRUCTOR:
    if s.SSN=i.SSN
        then
            s.ADDRESS=i.ADDRESS or
            (s ADDRESS null and i ADDRESS null)
```

- The available network database management systems usually do not support sophisticated userviews; they support only subschemas, which are trivial userviews. This means that normally the solution Union +Events should not be chosen in the design of a network schema.

2. **Convert every proper 1:m or m:m relation whose range is a concrete category** into a new abstract category and its two functional relations through a relation-split.

3. **Convert every proper many-to-many** relation into a category and two functional relations through a relation-split.

4. **Convert every m:1 nonattribute relation into a 1:m relation** by changing its direction and its name.

```
Example 6-23.
Instead of
  the-student relation from ENROLLMENT to STUDENT (m:1)
we have
  student-enrollment relation from STUDENT to ENROLLMENT (1:m)
```

5. **Convert every 1:m relation whose domain and range are the same category** into a new category, a 1:1 onto relation, and a 1:m onto relation, through a relation-split.

```
Example 6-24.
If we had:
  subdepartment relation from DEPARTMENT to DEPARTMENT (1:m)
then we would convert it into:
```
6. **Add the category** SYSTEM, which always has just one object — the enterprise or the world for which the database is being designed.

7. **Relate every independent category** C to the category SYSTEM:
   - system-C — relation from SYSTEM to C (1:m,onto)

8. If there are still abstract categories C that do not indirectly depend on SYSTEM (a very rare case), then relate them to SYSTEM:
   - system-C — relation from SYSTEM to C (1:m,onto)

Example 6-27.
There is no such problem in our schema. Let’s spoil our schema a bit to have such a problem. If the relation MAJOR-ST were onto:
   - major-st — relation from DEPARTMENT to STUDENT (1:m,onto)
and in addition we had a relation assigning one student council liaison to several departments:
   - student-council-liaison-for-department — relation from STUDENT to DEPARTMENT (1:m,onto)
then the categories STUDENT and DEPARTMENT would depend on each other. None of the categories would be independent. We have to link at least one of them to SYSTEM, so that they would indirectly depend on SYSTEM:
system-department → relation from SYSTEM to DEPARTMENT
(1:m,onto)

9. Define order-type of every nonattribute relation $R$ as follows:

If the domain of the relation is SYSTEM

a. then (‘order is by ascendance or descendance by fields’):

   (1) Let $C$ be the range of this relation. Pick an attribute of $C$, or a list of attributes, so that the application programs are most likely to access the objects of $C$ in the order which can be defined by these attributes.

   (2) Specify the order-type to preserve the ascendance-descendance of the above attributes.

   (A list of attributes establishes precedence between them: the objects are ordered primarily according to the first attribute in the list. Two objects that have the same value of the first attribute in the list are ordered according to the second attribute, and so on.)

   This need not be a deterministic (unambiguous) specification of the order — two distinct objects may have equal values in each of the attributes in the list. (In many DBMSs, such two objects are called duplicates.)

   The list of attributes may be empty, as it is, for example, when $C$ has no attributes at all.

Example 6-28.

- The order of SYSTEM-COURSE is ascending by NAME.
- The order of SYSTEM-QUARTER is ascending by YEAR, descending by SEASON. (Fortunately, the alphabetic order of seasons, ‘Winter’ > ‘Spring’ > ‘Fall’, coincides with their natural order.)
- The order of SYSTEM-INSTRUCTOR is ascending by LAST-NAME, ascending by FIRST-NAME.
- The order of SYSTEM-STUDENT is ascending by LAST-NAME, ascending by FIRST-NAME.
- The order of SYSTEM-DEPARTMENT has no constraint.

b. else (‘order is last’):

   Let the order be the order of insertion or connection, that is, there is a dynamic constraint (restriction) that when an object $x$ becomes connected by $R$ to an object $y$, this $x$ becomes the last by $\text{NEXT}_R$ among the objects connected by $R$ to $y$. 

Rishe-DDS p. 6.14
10. **Translate the integrity constraints** into the terms of the new schema:
   
   a. The constraints of the original schema.
   
   b. The additional constraints accumulated during the conversion process.

### 6.3. Network Languages

#### 6.3.1. Fourth-generation programming

Syntactically, this language is the same as the binary extension of Pascal but is used for network schemas. Pragmatically, there is one difference:

- In the Binary and Relational models, there is no order between the objects. Thus, the `for` loops are performed in an arbitrary order, transparent to the application programmer. The DBMS would usually attempt to find the most efficient order dependent on the circumstances of the program run and the physical structure of the database.

- In the Network Model, the loops are performed in the order specified by the `NEXT` ordering relations. This reduces the flexibility of the application program by making it depend on information which is not strictly relevant for the program’s goals. Also, this does not allow for optimization of the program by the DBMS.

---

**Example 6-29.**

The university has decided to expel all the students whose average grade is below 60 (out of 100). To prevent this wrong-doing to computer science students, the department offered a fictitious course, Computer-Pass, by Prof. Good, in which all computer science students are to receive a sufficient grade so as to not to be expelled, if possible.

The following program fabricates Prof. Good and the Computer-Pass course, enrolls students in this course, grades them accordingly, and prints the names of those computer science students whom this measure cannot help.

```plaintext
program Pass (Input, Output, UNIVERSITY-DB, UNIVERSITY-MASTER-VIEW);
var Computer-Pass-Course, Prof-Good, Good-Offer, comp-science,
    work, this-quarter, cs-student, her-enrollment, fictitious-enrollment:
    ABSTRACT;
var the-grade, desired-grade, number-of-grades,
    total-of-grades, current-year: INTEGER;
begin
(* Get the current year from the standard input file. *)
    read (current-year);
(* Fabricate the course, Prof. Good, and the offering. *)
    transaction begin

Rishe-DDS p. 6.15
```
create new Computer-Pass-Course in COURSE;
Computer-Pass-Course.NAME := ‘Computer Pass’;
create new Prof-Good in INSTRUCTOR;
Prof-Good.LAST-NAME := ‘Good’;
create new Good-Offer in OFFERING;
relate: Computer-Pass-Course COURSE-OFFERING Good-Offer;
relate: Prof-Good INSTRUCTOR-OFFERING Good-Offer;
for this-quarter in QUARTER
  where (this-quarter.YEAR = current-year and this-quarter.SEASON = ‘Winter’) do
    relate: this-quarter QUARTER-OFFERING Good-Offer;
end;

(* The following two nested loops will be performed only once. Inside the body of the
second loop, the variable comp-science will refer to the Computer Science
Department. *)

for computer-science-name in DEPARTMENT-NAMING
  where (computer-science-name.NAME = ‘COMPUTER SCIENCE’) do
    for comp-science in DEPARTMENT
      where (computer-science DEPARTMENT–DEPARTMENT-NAMING comp-science-name) do

      begin
      transaction begin
        create new work in WORK;
        relate: Prof-Good INSTRUCTOR-WORK work;
        relate: comp-science DEPARTMENT-WORK work
      end;
      for cs-student in STUDENT
        where (comp-science MAJOR-ST cs-student) do

        begin
        (* calculate this student’s current statistics: number-of-grades and total-of-
        grades *)
        number-of-grades := 0;
        total-of-grades := 0;
        for her-enrollment in ENROLLMENT

Rishe-DDS p. 6.16
where (cs-student STUDENT-ENROLLMENT her-enrollment and not her-enrollment FINAL-GRADE null) do

begin
the-grade :=
cs-student.FINAL-GRADE;
number-of-grades := number-of-grades + 1;
total-of-grades := total-of-grades + the-grade
end;

(* calculate the minimal desired grade in the computer-pass course, solving the equation *)

\[
\frac{total + x}{number + 1} = 60
\]

desired-grade := 60 * (number-of-grades + 1) – total-of-grades;

if desired-grade > 100

then

(* the student cannot be helped. Print a message *)

writeln (`The student ', cs-student.LAST-NAME, ` cannot be helped. Sorry!

else

if desired-grade > 60 then

transaction begin

create new fictitious-enrollment in ENROLLMENT;
relate: cs-student STUDENT-ENROLLMENT fictitious-enrollment;
relate: Good-Offer OFFERING-ENROLLMENT fictitious-enrollment;
fictitious-enrollment.FINAL-GRADE := desired-grade
end

end

end
6.3.2. Logic

Since every network schema is a binary schema, we can use the same language of Predicate Calculus as was defined for the Binary Model.

Example 6-30.
Has every student taken at least one course in 1990?

for every st in STUDENT:
    exists enrl in ENROLLMENT:
        ((st STUDENT-ENROLLMENT enrl) and
         exists offer in OFFERING:
             exists quarter in QUARTER:
                 quarter QUARTER-OFFERING offer and
                 offer OFFERING-ENROLLMENT enrl and
                 quarter.YEAR=1990)

Example 6-31.
Who took Prof. Smith’s courses?

get student.LAST-NAME where
    exists enrl in ENROLLMENT:
        exists prof in INSTRUCTOR:
            exists offer in OFFERING:
                prof.LAST-NAME='Smith' and
                prof INSTRUCTOR-OFFERING offer and
                offer OFFERING-ENROLLMENT enrl and
                student STUDENT-ENROLLMENT enrl

6.3.3. *Navigation

The language presented here is a lower-level extension of Pascal for network data manipulation. Unlike the structured extension of Pascal, here the user herself is responsible for the organization of loops and for navigating in the labyrinth of the database.

The model language presented in this section is an adaptation to Pascal of the logical features of the network data manipulation language proposed by the standard committee CODASYL/DBTG and used with minor variations in many commercial network database
management systems.

1. *Program heading* — same as in the structured extension.

```plaintext
Example 6-32.

program MY (input, output, University-data-base, University-principal-subschema)
```

2. There are *automatically-generated Pascal record types* for every record-type of the subschema.

```plaintext
Example 6-33.

type DEPARTMENT-NAMING =
    record
        the-name : String
    end;

type STUDENT =
    record
        last-name, first-name : String;
        birth-year : 1870..1990;
        address : String
    end;

type INSTRUCTOR =
    record
        last-name, first-name : String;
        birth-year : 1870..1990;
        address : String
    end;

type QUARTER =
    record
        year : 1980..1995;
        season : String
    end;

type COURSE =
    record
        name : String
    end;

type ENROLLMENT =
    record
        final-grade : 0..100
    end;

type NULL-RECORD =
    record
        end;
```

Rishe-DDS p. 6.19
3. **Automatically generated Pascal record variables** for every record-type in the subschema.

These variables bear the same names as the corresponding types.

These variables are called **template variables**. The template variable for a database record-type is used as a buffer to store the attributes of record-occurrences when they are moved between the database and the program.

**Example 6-34.**

```pascal
var student : STUDENT;
var enrollment : ENROLLMENT;
```

etc.

4. Pascal data type **ABSTRACT**.

The variables of type **ABSTRACT** reference database objects.

In many network database management systems this type is called **DBKEY**.

5. **System variables (read-only)**.

There are several automatically generated system variables. The user may not perform explicit assignments to these variables. These variables are updated by the system as a side effect of performing user commands.

**var Error-status :** (ok, end-of-set, error)

After the execution of any database command, this variable is automatically assigned one of the following values:

- **end-of-set** — if the user attempted to locate the next record occurrence in a particular set-occurrence and no next record-occurrence existed
- **error** — if another logical or physical error occurred

* Standard *Pascal* does not allow records without fields. So, a dummy field may have to be introduced:

```pascal
type DEPARTMENT =
record
  dummy : 0..0
end;
```
• **ok** — otherwise.

**var current-of-run-unit : ABSTRACT**

This variable references the last accessed object. Initially, it is the object SYSTEM.

**var current-of-record-type-record-type : ABSTRACT**

For every record-type in the subschema, there is a variable referencing the last accessed object of that record-type.

*Example 6-35.*

```pascal
var current-of-record-type-DEPARTMENT, current-of-record-type-STUDENT,
current-of-record-type-INSTRUCTOR, current-of-record-type-QUARTER,
current-of-record-type-COURSE, current-of-record-type-OFFERING,
current-of-record-type-ENROLLMENT, current-of-record-type-WORK,
current-of-record-type-DEPARTMENT-NAMING : ABSTRACT
```

**var current-of-set-type-set-type : ABSTRACT**

For every set-type in the subschema, there is a variable referencing the last accessed object of a record-type which is the owner or the member of this set-type. For the set-types whose owner is *SYSTEM*, these variables initially contain the object *SYSTEM*. Otherwise the variables are uninitialized.

*Example 6-36.*

```pascal
var current-of-set-type-SYSTEM-INSTRUCTOR, current-of-set-type-SYSTEM-STUDENT,
current-of-set-type-SYSTEM-DEPARTMENT, current-of-set-type-SYSTEM-COURSE,
current-of-set-type-SYSTEM-QUARTER, current-of-set-type-DEPARTMENT-WORK,
current-of-set-type-DEPARTMENT-DEPARTMENT-NAMING, current-of-set-type-MAJOR-ST,
current-of-set-type-MINOR-ST, current-of-set-type-INSTRUCTOR-OFFERING,
current-of-set-type-COURSE-OFFERING, current-of-set-type-OFFERING-OFFERING-ENROLLMENT,
current-of-set-type-STUDENT-ENROLLMENT : ABSTRACT
```

6. **Expressions** — there is *no* extension to the syntax of Pascal expressions.

Particularly, there are no operations on the objects of type ABSTRACT. For example, if $x$ is a variable of type ABSTRACT, and $A$ is an attribute, then $x.A$ would be an expression in the structured extension of Pascal but not in the navigational extension.

However, if $x$ is not an abstract variable, but a variable of a Pascal type **record**, for example

Rishe-DDS p. 6.21
\textbf{Statements}

7. \texttt{find first within set-name}

The first member object is located in the current set-occurrence of this set-type, the set occurrence to which the object \texttt{current-of-set-type-set-type} belongs.

If there are no member-objects in the set-occurrence, the system variable Error-status is set to \texttt{end-of-set}. (Otherwise, if the instruction is successfully performed, the variable Error-status is set to \texttt{ok}.)

If an object is found, the currency variables are updated: the found object becomes

- The current of run-unit
- The current of this set-type
- The current of this object’s record-type
- The current of every set-type whose owner or member is the category of this object

\begin{quote}
\textit{Example 6-37.}

Are there any students in the database?

\begin{verbatim}
find first within SYSTEM-STUDENT;
if Error-status = end-of-set
  then writeln (‘no’);
if Error-status = ok
  then writeln (‘yes’);
if Error-status = error
  then writeln (‘I do not know. I have system problems.’);
\end{verbatim}
\end{quote}

8. \texttt{get}

This instruction assigns the attributes’ values of the current object of run-unit to the template variable corresponding to the category of the object.

\begin{quote}
\textit{Example 6-38.}

Print the address of one student.
\end{quote}
(* locate one student, the one which happens to be the first in the order of SYSTEM-STUDENT *)

find first within SYSTEM-STUDENT;

(* assign the attribute values of the student to the variable student *)

get;

(* print *)

writeln ('The address of a student is:', student.ADDRESS)

9. find next within set-name

The next member object is located in the order of the set-type after the current object of the set-type.

If there are no next member-objects in the set-occurrence, the system variable Error-status is set to end-of-set.

If an object is found, the currency variables are updated: the found object becomes the current of run-unit, the current of this set-type, the current of this object's record-type, and the current of every set-type whose owner or member is the category of this object.

Example 6-39.

Print names and addresses of all the students.

find first within SYSTEM-STUDENT;

while (Error-status ≠ end-of-set) do

begin

(* assign the attribute values of the current student to the variable student *)

get;

(* print *)

writeln ('The address of Student', student.LAST-NAME, ' is: ', student.ADDRESS)

find next within SYSTEM-STUDENT

end;

10. find owner within set-name

The owner object of the set occurrence of the current object of the set-type is located.

The currency variables are updated: the found object becomes the current of run-unit, the current of this set-type, the current of this object’s record-type, and the current of every set-type whose owner or member is the category of this object.

Rishe-DDS p. 6.23
Example 6-40.
Print names and major departments of all the students.

```plaintext
find first within SYSTEM-STUDENT;
while (Error-status ≠ end-of-set) do
  begin
    (* assign the attribute values of the current student to the variable
      student *)
    get;
    (* find the major department of the student *)
    find owner within MAJOR-ST;
    (* find the name of the department; if the department has more than
     one name, the first found will do. *)
    find first within DEPARTMENT–DEPARTMENT-NAMING ;
    (* assign the attribute value of the current department-naming to the
     variable department-naming *)
    get
    (* print *)
    writeln (`The major department of Student', student.LAST-NAME,
     ` is: ', department-naming.NAME);
  find next within SYSTEM-STUDENT
  end;
```

11. **find db-key** is variable-of-type-ABSTRACT

(This statement is used to find and restore the currency of an object which was
previously accessed by the program and whose reference was saved in a variable.)

The object referred to by the variable is located. The currency variables are updated: the
found object becomes the current of run-unit and the current of every set-type whose
owner or member is the category of this object.

Example 6-41.
For every student, print the names of the students of the same major.

```plaintext
var student-to-remember: ABSTRACT;
begin
  find first within SYSTEM-STUDENT;
  while (Error-status ≠ end-of-set) do begin
```

Rishe-DDS p. 6.24
(* assign the attribute values of the current student to the variable *)

student *]

get;

(* find the major department of the student *)

find owner within MAJOR-ST;

(* print a heading for the current student’s list of co-majors. *)

writeln (‘Student’, student.LAST-NAME, ‘ has the same major as
the following students: ’);

(* remember the current student, so that we can return to her after we
have finished processing the co-majors. *)

student-to-remember := current-of-set-type-SYSTEM-
STUDENT;

(* process the students connected in the set-type MAJOR-ST to the
current department *)

find first within MAJOR-ST;

while (Error-status ≠ end-of-set) do

begin

get;

write (student.LAST-NAME);

find next within MAJOR-ST
end;

(* restore the currency of the student of the principal loop. *)

find db-key is student-to-remember;

find next within SYSTEM-STUDENT
end;

end.

12. find within set-name using attribute

Among the member objects in the set occurrence of the current object of the set-type,
find the first object whose value of the attribute is the same as in the template variable of
the member record-type.

The currency variables are updated: the found object becomes the current of run-unit, the
current of this set-type, the current of this object’s record-type, and the current of every
set-type whose owner or member is the category of this object.

This is roughly equivalent to the following program, where S is the set-type and rec is
the member record-type:

Rishe-DDS p. 6.25
attribute-to-compare := rec.attribute;

find first within $S$;

found := false;

while (Error-status ≠ end-of-set) and (not found) do
  begin
    get;
    found := (rec.attribute = attribute-to-compare);
    if not found
      then find next within $S$
  end;

Example 6-42.

How many times was the Databases course offered?

(* find the Databases course *)

course.NAME := 'Databases';

find within SYSTEM-COURSE using NAME;

(* count the offerings *)

number := 0;

find first within COURSE-OFFERING;

while Error-status ≠ end-of-set do
  begin
    number := number+1;
    find next within COURSE-OFFERING;
  end

writeln (‘Databases was offered ’, number, ‘ times.’)

13. modify record-type

The attributes’ values of the current object of this record-type are updated according to the values in the template variable of this record-type.

Example 6-43.

Change the name of every Fall quarter to Autumn.

find first within SYSTEM-QUARTER;

while (Error-status ≠ end-of-set) do
begin
if Error-status = error then begin
    writeln (` SYSTEM ERROR’); stop end;
get;
if quarter.SEASON = ‘Fall’ then
    begin
    quarter.SEASON := ‘Autumn’;
    modify QUARTER;
    end;
find next within SYSTEM-QUARTER
end;

14. erase record-type

The current object of the record-type is deleted from the database.
If the object is the owner of a set-occurrence of a set-type onto the member, then all the
member objects of that set occurrence are automatically erased. This process is
recursive: the deletion of some objects may trigger deletion of more objects.

Example 6-44.
Cancel everything that happened in any summer.
find first within SYSTEM-QUARTER;
while (Error-status ≠ end-of-set) do
    begin
    if Error-status = error then begin
        writeln (` SYSTEM ERROR’); stop end;
    get;
    if quarter.SEASON = ‘Summer’ then
        erase QUARTER;
    find next within SYSTEM-QUARTER
    end;

15. connect record-type to set-type

The current object of the record-type is inserted as a member into the current set-
occurrence of the set-type — the set-occurrence in which the current object of the set-
type is the owner or a member.

Example 6-45.
Let every student have the same minor as his major, assuming that every student had (until now) a major and no minor department.

```plaintext
find first within SYSTEM-STUDENT;
while (Error-status ≠ end-of-set) do
begin
  if Error-status = error then begin
    writeln (' SYSTEM ERROR'); stop end;
  get;
find owner within MAJOR-ST;
if Error-status = error then
  writeln (' Oh-oh. Wrong assumption about having a major
department for ', student.LAST-NAME);
connect STUDENT to MINOR-ST;
if Error-status = error then
  writeln (' Oh-oh. Wrong assumption about not having a minor
department for ', student.LAST-NAME);
find next within SYSTEM-STUDENT
end;
```

16. **disconnect record-type from set-type**
The current object of the record-type will no longer be a member in the current set-occurrence of the set-type (the set-occurrence in which the current object of the set-type is the owner or a member).

Example 6-46.
Let every student have the same minor as his major, by canceling the present minors. Assume that every student had a major department.

```plaintext
find first within SYSTEM-STUDENT;
while (Error-status ≠ end-of-set) do
begin
  if Error-status = error then begin
    writeln (' SYSTEM ERROR'); stop end;
  get;
```
disconnect STUDENT from MINOR-ST;
find owner within MAJOR-ST;
if Error-status = error then
  writeln (' Oh-oh. Wrong assumption about having a major
department for ', student.LAST-NAME);
connect STUDENT to MINOR-ST;
find next within SYSTEM-STUDENT
end;

17. reconnect record-type to set-type

The current object of the record-type will no longer be a member in its former set-
occurrence of the set-type.

Instead, the current object of the record-type is inserted as a member into the current
set-occurrence of the set-type (the set-occurrence in which the current object of the set-
type is the owner or a member).

Example 6-47.
Let every student have the same minor as his major, canceling the present minor,
assuming that every student had a major department.

find first within SYSTEM-STUDENT;
while (Error-status ≠ end-of-set) do
  begin
    if Error-status = error then begin
      writeln (' SYSTEM ERROR'); stop end;
    get;
    find owner within MAJOR-ST;
    if Error-status = error then
      writeln (' Oh-oh. Wrong assumption about having a major
department for ', student.LAST-NAME);
    reconnect STUDENT to MINOR-ST;
    find next within SYSTEM-STUDENT
  end;

18. store record-type
A new object of the *record-type* is created.
The object automatically gets the values of its attributes from the template variable of this record-type.
This object becomes the current of the *record-type*.

19. **transaction compound-statement**
— as in the structured extension of Pascal.

```pascal
Example 6-48.
Let Prof. Asteroid (a unique name for this instructor, so no confusion with other instructors is possible) offer the course *Databases* every quarter.

```
```
```
```
```pascal
instructor.LAST-NAME := 'Asteroid';
course.NAME := 'Databases';

find within SYSTEM-INSTRUCTOR using LAST-NAME;
find within SYSTEM-COURSE using NAME;
find first within SYSTEM-QUARTER;
while (Error-status ≠ end-of-set) do
begin
  if Error-status = error then begin
    writeln ('SYSTEM ERROR'); stop
  end;
get;
transaction begin
  store OFFERING;
  connect OFFERING to QUARTER-OFFERING;
  connect OFFERING to COURSE-OFFERING;
  connect OFFERING to INSTRUCTOR-OFFERING
end
find next within SYSTEM-QUARTER
end;
```

```
```
```
```
```pascal
```
```
```
```
```pascal
```
```
```
```pascal
Example 6-49.
A navigational program for the expulsion prevention problem which has been previously solved in the structured extension of Pascal.

Rishe-DDS p. 6.30
program Pass (Input, Output, UNIVERSITY-DB, UNIVERSITY-PRINCIPAL-SUBSCHEMA);

(* The comments in the boxes are the corresponding program fragments in the structured extension of Pascal. It is recommended that when a program needs to be written in the navigational language, it should first be written in the higher-level language. Then, when the higher-level program is translated into the lower-level navigational language, the commands of the original program should become algorithmic comments within the navigational program. *)

var Good-Offer: ABSTRACT;
var the-grade, desired-grade, number-of-grades, total-of-grades, current-year: INTEGER;
const null-year = 1870 (* to represent missing birth-year, assuming nobody was born in 1870*);
const null-name = ‘’ (* to represent missing names *);
const null-address = ‘’ (* to represent missing addresses *);
const null-grade = 0 (* to represent missing grades, assuming that 0 is never given as a real grade² *)
begin (* Get the current year from the standard input file. *)
  read (current-year);
  transaction begin
    (*
    create new Computer-Pass-Course in COURSE;
    Computer-Pass-Course.NAME := ‘Computer Pass’; *)
    course.NAME := ‘Computer Pass’;
    store COURSE;
    connect COURSE to SYSTEM-COURSE;
    (*
    create new instructor in INSTRUCTOR;
    instructor.LAST-NAME := ‘Good’; *)
    instructor.LAST-NAME := ‘Good’;
    instructor.FIRST-NAME := null-name;

² If the type of the variable enrollment.Final-grade technically allows for negative values, then a better representation of the missing grade is −1.
instructor.\emph{BIRTH-YEAR} ::= \texttt{null-year};
instructor.\emph{ADDRESS} ::= \texttt{null-address};

\texttt{store INSTRUCTOR;}

\texttt{connect INSTRUCTOR to SYSTEM-INSTRUCTOR;}

\begin{verbatim}
(*
  create new Good-Offer in OFFERING;
  relate: Computer-Pass-Course COURSE-OFFERING Good-Offer;
  relate: Prod-Good INSTRUCTOR-OFFERING Good-Offer; *)
\end{verbatim}

\texttt{store OFFERING;}

\texttt{connect OFFERING to INSTRUCTOR-OFFERING;}

\texttt{connect OFFERING to COURSE-OFFERING;}

\texttt{Good-Offer ::= current-of-record-type-OFFERING;}

\begin{verbatim}
(*
  for quarter in QUARTER
    where (quarter.YEAR = current-year \textbf{and} quarter.SEASON = ‘Winter’)
    do
    relate: quarter QUARTER-OFFERING Good-Offer;
  *)
\end{verbatim}

\texttt{found ::= \texttt{false};}

\texttt{find first within SYSTEM-QUARTER;}

\texttt{while (Error-status \neq \texttt{end-of-set} \textbf{and} \texttt{not} found) do}

\begin{verbatim}
  begin;
  if Error-status = error then begin
    writeln (‘ SYSTEM ERROR’); stop end;
  get;
  end;
  if not found then
    find next within SYSTEM-QUARTER
  end;
\end{verbatim}

\texttt{connect OFFERING to QUARTER-OFFERING;}

\texttt{end (* transaction*);}
(* Prepare-Department: *)

for department-naming in DEPARTMENT-NAMING
  where (department-naming.NAME = 'COMPUTER SCIENCE') do
    for department in DEPARTMENT
      where (department DEPARTMENT-DEPARTMENT-NAMING
             department-naming) do
      *

found := false;

find first within SYSTEM-DEPARTMENT;

while (Error-status ≠ end-of-set and not found) do
  begin;
    if Error-status = error then begin
      writeln ('SYSTEM ERROR'); stop end;
    get;
    found := false;

    find first within DEPARTMENT-DEPARTMENT-NAMING;

    while (Error-status ≠ end-of-set and not found) do
      begin;
        if Error-status = error then begin
          writeln ('SYSTEM ERROR'); stop end;
        get;
        found := (department-naming.NAME = 'Computer Science');
        if not found then
          find next within DEPARTMENT-DEPARTMENT-NAMING
        end;
        if not found then
          find next within SYSTEM-DEPARTMENT
        end;
  end;

(**end Prepare-department**) transaction begin

(*
  create new work in WORK;
  relate: instructor INSTRUCTOR-WORK work;
  relate: department DEPARTMENT-WORK work *)

Rishe-DDS p. 6.33
store WORK;
connect WORK to INSTRUCTOR-WORK;
connect WORK to DEPARTMENT-WORK;
end (* transaction *);

(* Student-loop: *)
for student in STUDENT
    where (department MAJOR-ST student) do *

find first within MAJOR-ST;
while (Error-status ≠ end-of-set) do
    begin
        if Error-status = error then begin
            writeln (` SYSTEM ERROR'); stop end;
        get;

        (**end of the opening of Student-loop **) (* calculate this student’s current statistics: number-of-grades and total-of-grades *)
        number-of-grades := 0;
        total-of-grades := 0;

        (* Enrollment-loop: *)
        for enrollment in ENROLLMENT
            where (student STUDENT-ENROLLMENT enrollment and not enrollment FINAL-GRADE null) do *

find first within STUDENT-ENROLLMENT;
while (Error-status ≠ end-of-set) do
    begin
        if Error-status = error then begin
            writeln (` SYSTEM ERROR'); stop end;
        get;
        if student.FINAL-GRADE ≠ null-grade then
            begin
                (**end of the opening of Enrollment-loop **) (* calculate this student’s current statistics: number-of-grades and total-of-grades *)
                the-grade := student.FINAL-GRADE;
                number-of-grades := number-of-grades + 1;
total-of-grades := total-of-grades + the-grade

(**closing Enrollment-loop**)  
end;

find next within STUDENT-ENROLLMENT  
end;

(**end Enrollment-loop**)  

(* calculate the minimal desired grade in the computer-pass course, solving the equation  
\( \frac{\text{total} + x}{\text{number} + 1} = 60 \) *)

desired-grade := 60 * (number-of-grades + 1) - total-of-grades;

if desired-grade > 100 then

(* the student cannot be helped. Print a message *)

writeln (`The student`, student.LAST-NAME, `cannot be helped. Sorry!`)  
else

if desired-grade > 60 then

transaction begin

(*
create new fictitious-enrollment in ENROLLMENT;
relate: student STUDENT-ENROLLMENT
fictitious-enrollment;
relate: Good-Offer OFFERING-ENROLLMENT
fictitious-enrollment;
fictitious-enrollment.FINAL-GRADE := desired-grade *)

enrollment.FINAL-GRADE := desired-grade;

store ENROLLMENT;

connect ENROLLMENT to STUDENT-ENROLLMENT;

find db-key is Good-Offer; (* We have to restore the currency of the fabricated offering remembered in the variable Good-Offer, because the currency may have changed in the meantime. *)

connect ENROLLMENT to OFFERING-ENROLLMENT;

end

(**closing Student-loop**)
Reference Schemas

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.
Figure Ref-1. A semantic schema for a university application.
Figure Ref-2. A relational schema for the university application.
7. FROM THE NETWORK TO THE HIERARCHICAL MODEL

This chapter defines the hierarchical data model and adapts the top-down database design methodology to hierarchical databases. Section 3 of this chapter discusses network database languages: application of the generic fourth-generation and logic-based languages.

The Hierarchical Model requires the schema and the instantaneous database to be trees. The Hierarchical Model is the oldest (albeit, not the best) major database model. However, it is still widely used in the industry.

This chapter presents a "modern view" of the hierarchical model. Some technicalities, particularly the clearly obsolete ones, are ignored.

7.1. Definitions

Hierarchical schema — a network schema such that for every category, excluding SYSTEM, there is exactly one relation coming into it from another category.

Example 7.1.

A hierarchical schema for the university application follows. Only the order-less part is shown in the figure.
Figure 7-1. An orderless hierarchical schema for the university application.

Note: We can prove the truth of the following statements from the definition of the hierarchical schemas:

a. The relations between the abstract categories of any orderless hierarchical schema form a set of directed trees. (This is the reason for the name *Hierarchical.*)

Rishe-DDS p. 7.2
b. The root of every tree is an independent category.

c. All the relations between distinct abstract categories are onto.

d. Since every category $C$ has only one entering relation $R_C$ from another category, every category has only one order $\text{NEXT}_{R_C}$. Thus, we can speak about the order of the objects of a category.

Example 7-2.
The following figures represent an instantaneous database for the schema of Figure ___. This instantaneous database represents the same state of the application’s real world as the binary instantaneous database of Figure ___ on page ____.

<table>
<thead>
<tr>
<th>id</th>
<th>last-name</th>
<th>first-name</th>
<th>birth-year</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>Brown</td>
<td>George</td>
<td>1956</td>
<td>112 Lucky Dr.</td>
</tr>
<tr>
<td>14352</td>
<td>Whatson</td>
<td>Mary</td>
<td>1953</td>
<td>231 Fortune Dr.</td>
</tr>
<tr>
<td>24453</td>
<td>Blue</td>
<td>John</td>
<td>1950</td>
<td>536 Orange Dr.</td>
</tr>
</tbody>
</table>

Figure 7-2. An instantaneous database for the hierarchical schema of the university application. Part I. The flat tree of INSTRUCTOR.
STUDENT

<table>
<thead>
<tr>
<th>last-name</th>
<th>first-name</th>
<th>birth-year</th>
<th>address</th>
<th>major-dept-</th>
<th>minor-dept-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Victory</td>
<td>Elizabeth</td>
<td>1966</td>
<td>100 Sun St.</td>
<td>Computer</td>
<td>Economics</td>
</tr>
</tbody>
</table>

COURSE ENROLLMENT

<table>
<thead>
<tr>
<th>instructor-id</th>
<th>course-name</th>
<th>year</th>
<th>season</th>
<th>final-grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>Gastronomy</td>
<td>1990</td>
<td>Fall</td>
<td>100</td>
</tr>
</tbody>
</table>

STUDENT

<table>
<thead>
<tr>
<th>last-name</th>
<th>first-name</th>
<th>birth-year</th>
<th>address</th>
<th>major-dept-</th>
<th>minor-dept-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Howards</td>
<td>Jane</td>
<td>1965</td>
<td>200 Dorms</td>
<td>Arts</td>
<td>Economics</td>
</tr>
</tbody>
</table>

COURSE ENROLLMENT

<table>
<thead>
<tr>
<th>instructor-id</th>
<th>course-name</th>
<th>year</th>
<th>season</th>
<th>final-grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>Gastronomy</td>
<td>1990</td>
<td>Fall</td>
<td>70</td>
</tr>
<tr>
<td>11332</td>
<td>Databases</td>
<td>1990</td>
<td>Fall</td>
<td>80</td>
</tr>
</tbody>
</table>

STUDENT

<table>
<thead>
<tr>
<th>last-name</th>
<th>first-name</th>
<th>birth-year</th>
<th>address</th>
<th>major-dept-</th>
<th>minor-dept-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wood</td>
<td>Michael</td>
<td>1964</td>
<td>110 Dorms</td>
<td>Arts</td>
<td>Economics</td>
</tr>
</tbody>
</table>

Figure 7-3. An instantaneous database for the hierarchical schema of the university application. Part II. The tree of STUDENT.
<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>main-name</th>
<th>DEPARTMENT NAMING</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science</td>
<td></td>
<td>CS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WORK</td>
<td>instructor-id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11332</td>
</tr>
<tr>
<td>Mathematics</td>
<td></td>
<td>Math</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WORK</td>
<td>instructor-id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11332</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>24453</td>
</tr>
<tr>
<td>Physics</td>
<td></td>
<td></td>
<td>14352</td>
</tr>
<tr>
<td>Arts Economics</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7-4.** An instantaneous database for the hierarchical schema of the university application. Part III. The tree of *DEPARTMENT*.
## COURSE

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Databases</td>
</tr>
</tbody>
</table>

### COURSE OFFERING

<table>
<thead>
<tr>
<th>instructor-id</th>
<th>year</th>
<th>season</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>1990</td>
<td>Fall</td>
</tr>
</tbody>
</table>

## COURSE

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Football</td>
</tr>
</tbody>
</table>

### COURSE OFFERING

<table>
<thead>
<tr>
<th>instructor-id</th>
<th>year</th>
<th>season</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>1990</td>
<td>Fall</td>
</tr>
</tbody>
</table>

## COURSE

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gastronomy</td>
</tr>
</tbody>
</table>

### COURSE OFFERING

<table>
<thead>
<tr>
<th>instructor-id</th>
<th>year</th>
<th>season</th>
</tr>
</thead>
<tbody>
<tr>
<td>11332</td>
<td>1990</td>
<td>Fall</td>
</tr>
</tbody>
</table>

**Figure 7-5.** An instantaneous database for the hierarchical schema of the university application. Part IV. The tree of `COURSE`.  

Rishe-DDS p. 7.6
Figure 7-6. An instantaneous database for the hierarchical schema of the university application. Part V. The flat tree of QUARTER.

Hierarchical schema terminology

Segment type — record type.

Example 7-3.
DEPARTMENT.

Segment occurrence — record occurrence.

Example 7-4.
One course and its name.

Example 7-5.
One student and his or her last name, first name, address, and the main names of his or her major and minor departments.

If there is a relation from a segment-type $C_1$ to another $C_2$, then $C_1$ is the parent of $C_2$ and $C_2$ is a child of $C_1$.

Example 7-6.
DEPARTMENT is the parent of DEPARTMENT-NAMING.
DEPARTMENT-NAMING is a child of DEPARTMENT.
Figure 7-7. Family relations in a hierarchical schema.

Root segment type — a segment type whose parent is SYSTEM.

Example 7-7.

DEPARTMENT, INSTRUCTOR, STUDENT, COURSE, QUARTER.

The hierarchical model is not the most natural model to describe the real world of an application. One should notice that what is sometimes called "a hierarchical real world" usually implies a hierarchical relation between objects in the real world, not a hierarchical schema.

Example 7-8.

Consider an oversimplified military world, where every soldier has at most one commander. A binary schema for this world is:
7.2. Database Design

Conversion algorithm of a semantic schema into a hierarchical schema whose quality is among the highest possible for the latter (provided the original semantic schema is of high quality).

1. Convert the binary schema into a network schema.
2. For every abstract category, excluding SYSTEM, choose its parent-relation R so that

   When all the relations, except the parent relations, are removed from the schema, every abstract category would still indirectly depend on SYSTEM.

Note: The parent relation for category C must be onto C.

Example 7-9.

The parent relation of OFFERING may be COURSE-OFFERING or INSTRUCTOR-OFFERING or QUARTER-OFFERING.

The parent relation for ENROLLMENT may be OFFERING-ENROLLMENT or STUDENT-ENROLLMENT.

The parent relation for WORK may be DEPARTMENT-WORK or INSTRUCTOR-WORK.

The parent relation for DEPARTMENT-NAMING is DEPARTMENT–DEPARTMENT-NAMING.

Note: The parent relations of independent categories are from SYSTEM.

Example 7-10.

The parent relation of STUDENT is SYSTEM-STUDENT.

The parent relation of INSTRUCTOR is SYSTEM-INSTRUCTOR.

The parent relation of DEPARTMENT is SYSTEM-DEPARTMENT.
The parent relation of \textit{COURSE} is \textit{SYSTEM-COURSE}.
The parent relation of \textit{QUARTER} is \textit{SYSTEM-QUARTER}.

3. Convert the schema into a tree: for every abstract category \( C \) that has more than one entering relation, do the following.

Let \( R \) be its parent relation and \( R_1, \ldots, R_n \) the other entering relations, whose domains are \( C_1, \ldots, C_n \).

For each of the relations \( R_i \) perform the following:

\( a. \) Find a key for the category \( C_i \). (\( C_i \) is the domain of the relation \( R_i \) which we intend to eliminate.) Add the key to the schema (if this key has not been added yet).

Rishe-DDS p. 7.10
Add to $C$ a new attribute, or a group of attributes, that is the composition of the inverse of $R_i$ on the key of $C_i$.

Remove the relation $R_i$ (and its order).
Add the referential integrity constraints that the values of the new attributes of $C$ must refer to the values of the keys of the categories $C_i$. (This is similar to what is done in the relational model.)

4. Remove from every category attributes which can be inferred from other attributes of that category, its parent, or its ancestors.

5. **Translate the integrity constraints** into the terms of the new schema:
   
   - The constraints of the original schema.
   - The additional constraints accumulated during the conversion process.

**Note:**

- In the first step, it is not strictly necessary to go all the way through in the conversion from the binary schema to the network schema. We can omit the substep where the 1:m relations from a category to itself are eliminated. Later, in Step 3 of the hierarchical conversion, these relations will be taken care of as all the nonparent relations.

If we do this in Step 3 rather than in Step 1 we get a slightly better hierarchical schema. If we were to follow the network conversion in Step 1 fully, we could get one extra category for the relation-split of each 1:m relation from a category to itself.
7.3. Hierarchical Languages

7.3.1. Fourth-generation programming

Syntactically, this language is the same as the binary extension of Pascal but is used for the hierarchical schema (every hierarchical schema is a binary schema). Pragmatically, there is one difference: the \texttt{for} loops are performed in the order corresponding to the ordering of objects in the database. (In the Binary Model, the order in which a \texttt{for} loop is performed is transparent to the application programmer.)

\begin{example}

The university has decided to expel all the students whose average grade is below 60 (out of 100). To prevent this wrong-doing to computer science students, the department offered a fictitious course, Computer-Pass, by Prof. Good, in which all computer science students are to receive a sufficient grade so as to not to be expelled, if possible.

The following program fabricates Prof. Good and the Computer-Pass course, enrolls students in this course, grades them accordingly, and prints the names of those computer science students whom this measure cannot help.

\begin{verbatim}
program Pass (Input, Output, UNIVERSITY-DB, UNIVERSITY-MASTER-VIEW);
var Computer-Pass-Course, Prof-Good, Good-Offer, comp-science, comp-science-name, Good-employment, cs-student, her-enrollment, fictitious-enrollment: ABSTRACT;
var the-grade, desired-grade, number-of-grades, total-of-grades, current-year: INTEGER;

begin (* Get the current year from the standard input file. *)
read (current-year);

transaction begin
   create new Computer-Pass-Course in COURSE;
   Computer-Pass-Course.NAME := `Computer Pass’;
   create new Prof-Good in INSTRUCTOR;
   Prof-Good.LAST-NAME := `Good’;
   Prof-Good.ID := 1234;
   create new Good-Offer in OFFERING;
   relate: Computer-Pass-Course OFFERING Good-Offer;
   Good-Offer.INSTRUCTOR-ID := 1234;
   Good-Offer.YEAR := current-year;
   Good-Offer.SEASON := `Winter’;
\end{verbatim}
\end{example}
for comp-science-name in \textit{DEPARTMENT-NAMING}
  where (comp-science-name.\textit{NAME}= 'COMPUTER SCIENCE') do
for comp-science in \textit{DEPARTMENT}
  where (comp-science \textit{DEPARTMENT–DEPARTMENT-NAME} comp-science-name) do begin
\textbf{transaction begin}
  create new \textit{Good-employment} in \textit{WORK};
  relate: comp-science \textit{DEPARTMENT–WORK} \textit{Good-employment};
  \textit{Good-employment.ID} := 1234
\textbf{end};
for cs-student in \textit{STUDENT}
  where (cs-student.\textit{MAJOR-DEPARTMENT-MAIN-NAME} = comp-science.\textit{MAIN-NAME}) do begin
(* calculate this student’s current statistics: number-of-grades and total-of-grades *)
  number-of-grades := 0;
  total-of-grades := 0;
for her-enrollment in \textit{ENROLLMENT}
  where (cs-student \textit{ENROLLMENT} her-enrollment \textbf{and not} her-enrollment \textit{FINAL-GRADE null}) do begin
the-grade :=
  her-enrollment.\textit{FINAL-GRADE};
  number-of-grades :=
  number-of-grades + 1;
  total-of-grades := total-of-grades + the-grade
\textbf{end};
(* calculate the minimal desired grade in the computer-pass course, solving the equation \((total + x)/(number + 1) = 60 \) *)
  desired-grade := 60 * (number-of-grades + 1) — total-of-grades;
if desired-grade > 100
  then
    (* the student cannot be helped. Print a message *)
writeln (`The student ', cs-student.LAST-NAME, ` cannot be helped. Sorry!')

else if desired-grade > 60 then

   transaction begin
     create new fictitious-enrollment in ENROLLMENT;
     relate: cs-student ENROLLMENT fictitious-enrollment;
     fictitious-enrollment.FINAL-GRADE := desired-grade;
     fictitious-enrollment.YEAR := current-year;
     fictitious-enrollment.SEASON := `Winter';
     fictitious-enrollment.INSTRUCTOR-ID := 1234;
     fictitious-enrollment.COURSE-NAME := `Computer Pass'
   end

end

end.

7.3.2. Logic

Since every hierarchical schema is a binary schema, we can use the same language of Predicate Calculus as was defined for the Semantic Binary Model.

Example 7-12.
Has every student taken at least one course in 1990?

   for every st in STUDENT:
     exists enrl in ENROLLMENT:
       ((st STUDENT-ENROLLMENT enrl ) and
        (enrl.YEAR=1990))

Example 7-13.
Who took Prof. Smith’s courses?

   get student.LAST-NAME where
     exists enrl in ENROLLMENT:
       (student STUDENT-ENROLLMENT enrl and

Rishe-DDS p. 7.15
exists inst in INSTRUCTOR:
   (inst.LAST-NAME = 'Smith' and
    enrl.INSTRUCTOR-ID = inst.ID))

Example 7-14.
What students have their average grade below 60?

   get std.LAST-NAME
   where 60 >
      average enrl.FINAL-GRADE
      where
         std STUDENT-ENROLLMENT enrl

Example 7-15.
Print the average of grades for every computer science student.

   get student.LAST-NAME,
      (average enrollment.FINAL-GRADE
       where
         student STUDENT-ENROLLMENT enrollment)
   where student is a STUDENT and
      (student.MAJOR-DEPARTMENT-MAIN-NAME = 'Computer Science')

Example 7-16.
How many students are there in the university?

   get (count std where std is a STUDENT)

Reference Schemas

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.
Figure Ref-1. A semantic schema for a university application.
**Figure Ref-2.** A relational schema for the university application.
8. THE PROGRESSION OF DATABASE MODELS

Four database models have been discussed in this text: Semantic, Relational, Network, and Hierarchical. This chapter compares aspects of theses models.

Each of the models provides a certain degree of data independence — the isolation of the application programmer from representational or implementational details. The following table illustrates five levels of data independence.

<table>
<thead>
<tr>
<th>Level</th>
<th>What is transparent to the user?</th>
<th>Database model</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>The representation of information by data</td>
<td>Semantic Binary</td>
</tr>
<tr>
<td>IV</td>
<td>The organization of access to data</td>
<td>Relational</td>
</tr>
<tr>
<td>III</td>
<td>The physical implementation of the logical data-access structure</td>
<td>Network</td>
</tr>
<tr>
<td>II</td>
<td>The physical implementation of the special data structure Hierarchy</td>
<td>Hierarchical</td>
</tr>
<tr>
<td>I</td>
<td>The physical implementation of logical files by bytes on disks</td>
<td>File management systems</td>
</tr>
</tbody>
</table>

Figure 8-1. Levels of data independence.

The Relational, Network, and Hierarchical models were derived from the Binary Model as its subsets. A schema in any one of those models is a binary schema satisfying certain criteria. Those criteria are related to the implementational restrictions of the model. The following diagram shows the subsets of the Binary Model.
Figure 8-2. The generalization of data models. For example, the relational schemas form a subcategory of the binary schemas.

The following figure depicts the schema design methodologies presented in this text. The goal of the methodologies is to produce high-quality databases in the Relational, Network, and Hierarchical models.
To review and compare some of the structural characteristics of the four database models studied in this text, the following series of examples solves the same query in the Predicate Calculus language, using, in turn, each of the four database models.

The examples use the four reference schemas of the university application at the end of this book. The query prints the pairs of the names of students and instructors, where the instructor works in the student’s major department.

**Example 8-1.**

**Binary:**

get s.LAST-NAME, i.LAST-NAME

where i WORKS-IN s.MAJOR-DEPARTMENT
Example 8-2.

**Relational:**

get s.LAST-NAME, i.LAST-NAME

where

i is an INSTRUCTOR and
s is a STUDENT and
exists w in WORK:

(w.INSTRUCTOR-ID-in-key = i.ID-key and
s.MAJOR-DEPARTMENT-MAIN-NAME =
w.DEPARTMENT-MAIN-NAME-in-key)

Example 8-3.

**Network:**

get s.LAST-NAME, i.LAST-NAME

where

i is an INSTRUCTOR and
s is a STUDENT and
exists w in WORK:

(i INSTRUCTOR-WORK w and
exists d in DEPARTMENT:

(d DEPARTMENT-WORK w and
  d MAJOR-ST s))

Example 8-4.

**Hierarchical:**

get s.LAST-NAME, i.LAST-NAME

where

i is an INSTRUCTOR and
s is a STUDENT and
exists w in WORK:

(i.ID = w.INSTRUCTOR-ID and
exists d in DEPARTMENT:

Rishe-DDS p. 8.4
Reference Schemas

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.
Figure Ref-1. A semantic schema for a university application.
**Figure Ref-2.** A relational schema for the university application.