The LIBD Tutorial Series

Introduction to

Database Design

Fifth Edition - March 2002

LIBD - Laboratory of Database Application Engineering
The University of Namur - Institut d'Informatique

Credits

This series of tutorials is a result of the Knowledge & Technology transfer action of the LIBD (Laboratory of Database Application Engineering). The LIBD is (and has been) supported by:

l'Université de Namur (FUNDP)
la Communauté Française de Belgique
la Région Wallonne
l'Union Européen
and

by a consortium of companies and public administrations comprising:

ACEC-OSI, AGD, ARIANE-II, ASCII, Banque UCL (Fortis), BBL, Carrières du Hainaut, Centre de Recherche Public H. Tudor, Clinique Universitaires St-Luc, Cockerill-Sambre, CONCIS, Daimler-Chrysler, DIGITAL, D'Ieteren, EDF, EPFL, Euro View Services, Fortis-CGER, Groupe S, IBM, Institut National de Criminalistique.Ministère de la Région Bruxelles-Capitale, OBLOG Software, ORIGIN, Régie des bâtiments, TEC Charleroi, Ville de Namur, Winterthur, 3 Suisses.

Contacts

Professor Jean-Luc Hainaut
University of Namur - Institut d'Informatique
rue Grandgagnage, 21 • B-5000 Namur (Belgium)
jlhainaut@info.fundp.ac.be - http://www.info.fundp.ac.be/libd

Table of contents

Table of contents

Introduction

1. Building our first database

1.1 Introduction 1-2
1.2 Creating a new project 1-3
1.3 Defining a new schema 1-5
1.4 Defining entity types COMPANY and PRODUCT 1-7
1.5 Entering entity type attributes 1-8
1.6 Entering relationship type MANUFACTURES 1-10
1.7 Defining entity type identifiers 1-12
1.8 Documenting the schema 1-12
1.9 Producing a SQL database 1-14
1.10 Saving the project 1-16
1.11 Quitting DB-MAIN 1-17
Summary of Lesson 1 1-18
Exercises for Lesson 1 1-19

2. A closer look at schemas

2.1 Starting Lesson 2 2-2
2.2 On including database schemas into a document 2-2
2.3 Graphical views of a schema 2-3
2.4 Textual views of a schema 2-6
2.5 Manipulating the graphical components of a schema 2-10
2.6 Navigation through textual views 2-15
2.7 Reordering attributes and roles 2-16
2.8 Generating reports 2-18
2.9 Copying objects 2-20
2.10 Pasting notes 2-21
2.11 Quitting the lesson 2-21
Summary of Lesson 2 2-22
Exercises for Lesson 2 2-24

3. Multi-product projects

3.1 Starting Lesson 3 3-1

3.2 Conceptual and logical schemas 3-1
3.3 SQL code generation 3-5
3.4 Generating reports 3-8
3.5 Multi-product project 3-8
3.6 Deleting objects 3-10
3.7 Quitting the lesson 3-11
Summary of Lesson 3 3-12
Exercises for Lesson 3 3-12

4. Conceptual Modeling

4.1 Starting Lesson 4 4-2
4.2 Updating an object 4-2
4.3 What is a conceptual schema? 4-2
4.4 Cardinality of an attribute 4-4
4.5 Mandatory and optional attributes 4-5
4.6 Single- and multivalued attributes 4-5
4.7 Atomic and compound attributes 4-5
4.8 Multiple identifiers 4-6
4.9 Hybrid identifiers 4-7
4.10 N-ary relationship types 4-9
4.11 Relationship types with attributes 4-9
4.12 Relationship types with identifier(s) 4-11
4.13 Cyclic relationship types 4-13
4.14 The complete schema 4-16
4.15 Quitting the lesson 4-16
Summary of Lesson 4 4-17
Exercises for Lesson 4 4-18

5. Logical and Physical Modeling

5.1 Starting Lesson 5 5-2
5.2 What is a logical schema? 5-2
5.3 Transformation into a logical schema 5-3
5.4 Reference attributes (foreign keys) 5-6
5.5 Equality reference 5-8
5.6 Defining a foreign key 5-9
5.7 Access keys 5-10
5.8 Defining entity collections 5-13
5.9 Name processing 5-15
5.10 SQL code generation 5-18
5.11 Quitting the lesson 5-21
Summary of Lesson 5 5-22

Exercises for Lesson 5 5-23

6. Advanced Conceptual Modeling

6.1 Starting Lesson 6 6-2
6.2 Subtypes and supertypes (is-a relations) 6-2
6.3 Properties of the subtypes of an entity type 6-5
6.4 Supertype / subtype inheritance 6-8
6.5 Coexistent components of an entity type 6-11
6.6 Schema transformations : a first glance 6-14
6.7 Exclusive components of an entity type 6-17
6.8 Groups with at least one, or exactly one, existing component 6-19
6.9 Quitting the lesson 6-21
Summary of Lesson 6 6-22
Exercises for Lesson 6 6-23

7. Conceptual Analysis (1)

7.1 Objective of these lessons 7-2
7.2 Conceptual analysis and design 7-2
7.3 The case study 7-3
7.4 The analysis 7-3
7.5 Starting Lesson 7 7-5
7.6 Starting the analysis 7-5
7.7 The books 7-5
7.8 The copies 7-9
7.9 The authors 7-14
7.10 The current schema 7-17
7.11 Quitting the lesson 7-17
Technical addendum 7-19
7.12 The attribute/entity type transformation 7-19
Summary of Lesson 7 7-27
Exercises for Lesson 7 7-27

8. Conceptual Analysis (2)

8.1 Starting Lesson 8 8-2
8.2 The analysis 8-2
8.3 The borrowers 8-2
8.4 Borrowings and projects 8-7
8.5 Borrowing history 8-9
8.6 The final schema 8-12
8.7 Quitting the lesson 8-15

Technical addendum 8-16
8.8 Discussion on the attribute/entity type transformation (continued) 8-16
Summary of Lesson 8 8-19
Exercises for Lesson 8 8-20

9. Logical Design

9.1 Starting Lesson 9 9-2
9.2 Logical design 9-2
9.3 The concept of Relational Logical Schema 9-3
9.4 Transformational approach to Logical design 9-6
9.5 Dealing with one-to-many relationship types 9-8
9.6 Processing many-to-many relationship types 9-11
9.7 Transforming complex relationship types 9-13
9.8 Logical design, at last! 9-15
9.9 Quitting the lesson 9-25
Technical addenda 9-27
9.10 On the rel-type/entity type transformation 9-27
9.11 On the rel-type/reference attribute transformation 9-31
9.12 On the technical ID transformation 9-38
Summary of Lesson 9 9-40
Exercises for Lesson 9 9-41

10. Logical Design (2)

10.1 Starting Lesson 10 10-2
10.2 What to do next? 10-2
10.3 Transforming the compound attributes 10-2
10.4 Transforming the multivalued attributes 10-4
10.5 An (almost) SQL-compliant schema 10-9
10.6 The names 10-12
10.7 Quitting the lesson 10-12
Technical addenda 10-13
10.8 On the equivalence of Instance and Value representations 10-13
10.9 On transforming compound attributes 10-14
Summary of Lesson 10 10-18
Exercises for Lesson 10 10-18

11. Logical Design (3)

11.1 Starting Lesson 11 11-2
11.2 Working more systematically 11-2
11.3 Transforming the IS-A relations 11-4

11.4 A transformation plan 11-6
11.5 The Global transformation Assistant 11-14
11.6 Quitting the lesson 11-21
Technical addenda 11-22
11.7 IS-A transformation revisited 11-22
11.8 Elementary schema analysis 11-28
11.9 Advanced schema analysis 11-29
11.10 Advanced schema transformation 11-37
Summary of Lesson 11 11-43
Exercises for Lesson 11 11-44

12. Physical design

12.1 Starting Lesson 12 12-2
12.2 What is a physical schema? 12-2
12.3 And what about physical design? 12-3
12.4 Building the physical schema of a database 12-4
12.5 Redundant access keys 12-6
12.6 The TECH descriptions 12-11
12.7 Generating the DDL schema 12-11
12.8 Getting help from DB-MAIN 12-15
12.9 Quitting the lesson 12-17
Summary of Lesson 12 12-18
Exercises for Lesson 12 12-19

References

Index

Introduction

Database Design

Database design is a part of the Software Engineering domain, through which application developers specify, build and maintain large programs. More particular, database design is the art of drawing, validating and implementing correct and efficient permanent data structures, i.e., files and databases.

Transforming an art into a science, or at least into a discipline, is not an easy task, especially in software engineering. On the one hand, engineering requires a coordinated set of models, techniques, methods and tools, the development of which is far from obvious. On the other hand, practitioners have to be convinced that a disciplined approach to application design brings major benefits in the long term. Which is no easy task either!

Database models, techniques, methods and tools

The *database models* will be used to specify information/data structures at different levels of abstraction. They must offer an easy and intuitive way to state that *customers place orders*, *products are identified by their Product Number* or that *an order is placed by one customer* only. They must also make it possible to describe more technical aspects such as *CUSTOMER records are stored in the file CUST.DAT*, or *the index IDX_CITY is associated with the table PERSONNEL*.

Database engineering techniques encompass schema manipulation operators that are intended to improve the quality and the efficiency of the data structures. Normalization, validation, optimization, schema transformations and coding patterns are some of the most important techniques.

Database methods organize the whole work of building an actual database from the users requirements. They specify which tasks must be performed, in which order, and with which criteria in mind. They heavily rely on sophisticated *techniques* and produce documents expressed in database *models*.

Building large databases (say, from 200 to 20,000 tables) cannot be carried out without the support of powerful *CASE tools* (for Computer-Aided Software Engineering) that help the developer in applying the database design method and techniques. For instance, the mere SQL-DDL code that builds the database structures can span several thousands pages.

Database engineering education

A database is a piece of art, and, according to many designers, *carving* a database is a matter of experience, of feeling and of the personal temperament and taste of the artist. On the other hand, the requirements of this discipline often are overlooked by unexperienced developers. Indeed, building a 3-table database is not that difficult. Adding, from time to time, a table or two according to the needs of the program being developed is quite easy too. This incremental approach, as can be guessed, most often results into an awkward database structure that will prove unable to adapt to the evolving requirements of today information systems, and that will lead to poor performance.

Hence the importance of database engineering education, not only in the schools and universities, but also among active practitioners.

Database Engineering

Designing a database is just the beginning of the story: *maintaining* a database, transforming it according to new organization requirements (*evolution*), redocumenting a legacy database (*reverse engineering*), porting it to a new platform according a new architecture (*reenginering*), *integrating* independent databases, *federating* existing databases, *migrating* data from a format to another, coping with *spatial* and *temporal* aspects of data, are other major processes that deserve being addressed in a disciplined way. Database engineering is a large domain that must rely of powerful models, techniques, methods and tools, that go beyond mere database design.

The DB-MAIN project

DB-MAIN is a major research programme of the LIBD since 1993. The very objective of this long term project is the development of models, methods, CASE tools and educational materials that should help building, maintaining and reengineering complex, evolving, data-intensive applications.

One of the main result of the programme is the DB-MAIN CASE tool.

The technology transfer aspects of LIBD

As an academic institution, the University of Namur, and particularly the Institute of Informatics and the LIBD, are strongly committed to making knowledge available to as large as possible an audience. Accordingly, most results of the research projects are translated into educational materials such as case studies, lectures and training seminars, mainly intended to the students

of the university and to the industry. This document is one of the products that find their place in the technology transfer results of the LIBD.

About this document

This tutorial aims to introduce the reader to database engineering problems and processes by developing a small database step-by-step. Though the first lessons may appear primarily as a user's guide for the DB-MAIN tool, this book basically is a learning-by-doing attempt to tackle some of the most important problems and reasoning encountered when designing and implementing a database through a disciplined approach. Coping with these problems through the use of a CASE tool mainly is a way to familiarize the reader with these problems in a (hopefully) more attractive way. However, as a side effect, it will also introduce to the use of a powerful and original development environment that can solve complex problems that generally are out of the scope of most current CASE tools.

This tutorial is sliced into graduated lessons that go from the basics to more advanced topics. Each lesson is accompanied by suggested exercices. A reader in good physical and mental condition should not spend more than 60 minutes on each lesson.

Warning

These lessons are no substitute for the more technical documents of the programme. In particular, mastering topics such as normalization, transformations, optimization, methodology modeling, reverse engineering, maintenance or CASE programming will require a more in-depth treatment that will be addressed in specific documents and materials. When needed, the lessons will refer to these documents.

How to start?

The best way to start this tutorial is to spend some time (no more than one hour) walking through a very small document called *1st-Step*. This micro-tutorial is intended to introduce the reader to the very basics of database analysis and development, and to the main operations of the DB-MAIN CASE tool. This document is available as a Microsoft help file (*1st-step.hlp*). It can be used as an independent document, but it can be opened from the welcome panel of the DB-MAIN tool (large button "First Steps"). A PDF version also is available (*1st-Step.pdf*).

What to do next?

A more technical tutorial is being written with the title *Computer-Aided Database Engineering - Volume 1: Database Models*. Its goal is to help the reader to master the basic and advanced concepts of the data model that has been developed in the LIBD, and to learn how to use it through the DB-MAIN tool.

Where to find the educational materials?

The documents and software mentioned in this introduction as well as other, generally more advanced, documents, can be obtained from the site of the laboratory:

htp://www.info.fundp.ac.be/libd

Most of them can be freely downloaded and used for education purpose.

Lesson 1

Building our first database

Objective

In this first lesson, the reader will learn how to start and quit the DB-MAIN CASE tool, how to introduce a simple Entity-Relationship conceptual schema, and how to translate it into table and column structures expressed into the SQL language. S/he will also save her/his work for further use.
Above all, the reader will get an insight into what *Database Design* is all about.

Preliminary checking

For this lesson, be sure that the DB_MAIN directory includes the DB_MAIN.EXE program (the CASE tool) as well as all the run-time libraries (`*.dll`). See the `README.TXT` file for further detail.

This lesson assumes that you use DB-MAIN Version 5, but is valid for other versions as well.

1.1 Introduction

We will develop a very simple database intended to describe companies that manufacture products. Through this process we will familiarize ourselves with some important concepts in database engineering.

For instance, we will learn that besides the data structures that are built in the computer, and in which we will store the data about these *companies which manufacture these products*, there exists another, more abstract and more intuitive way to describe these concepts, namely the **conceptual schema**. While data are stored into tables or into files, a conceptual schema describes the concepts in terms of entity types (classes of similar objects), attributes (entity properties) and relationship types (associations holding among entities).

The most straightforward **conceptual schema** comprises the entity type `COMPANY`, which describes the class of companies, and the entity type `PRODUCT`, representing the class of products. The fact that companies manufacture products is represented by a *many-to-one* relationship type called `manufactures` connecting their entity types. We will give these entity types some attributes that describe the properties of the companies (such as their company identifier, their name and their revenue) and of the products.

1.2 Starting DB-MAIN

Through the Explorer (or File Manager), we go into the DB_MAIN directory, and we start the DB_MAIN program by double-clicking on the `DB_MAIN.EXE` name or on the `DB-MAIN` icon. We acknowledge the presentation box by clicking on the OK button, or by pressing the `Enter` key. The main DB-MAIN window appears, showing, among others, the *Menu bar* (with two items only: **File** and **Help**), the *Tool bar* (with a few buttons, among which

are *build a new project* and *open an existing project*), the *Workspace*, in which the project window will be displayed (currently empty), and the *Status bar*.

Figure 1.1 - The main window of DB-MAIN.

1.3 Creating a new project

We are ready to open a new project through the command **File / New project**. This command opens a *Project Property box* (or *Project box* for short), which asks us some information about the new project. Our project will be called `MANU-1` and will be given the short name `M1`. We validate the operation by clicking on button `OK`.

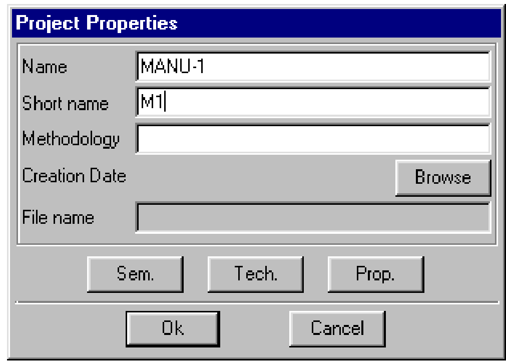


Figure 1.2 - The properties of the new project.

Note 1. There is a simpler way to open a new project, namely by pressing the `New project` button in the Tool bar.■

Now, a new window, namely the *Project window*, appears in the DB-MAIN workspace. Currently, it includes a small rectangle, which is the iconic representation of the project itself (any DB-MAIN object has a graphical representation). To examine its properties, try **File** / **Project Properties**[1]. Later on, this window will also show all the products of the project, such as the various schemas and texts, together with their relationships[2].

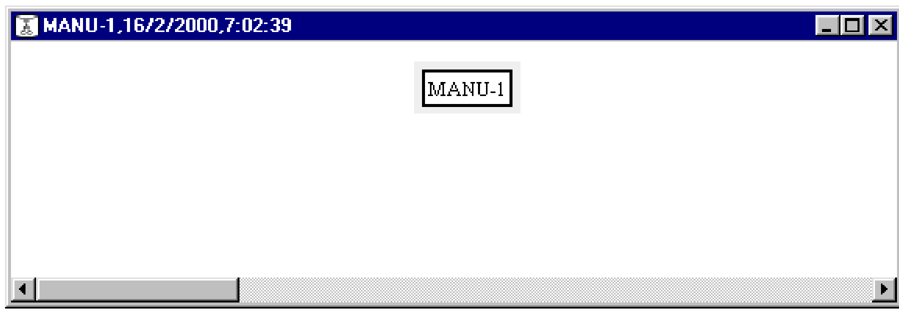


Figure 1.3 - The project window in which all the documents of the project will appear.

The Menu bar and the Tool bar have changed too, offering more functions that will be used later on. Make sure that the *Standard tools bar* is available. Otherwise use **Windows** / **Standard tools** to make it visible.

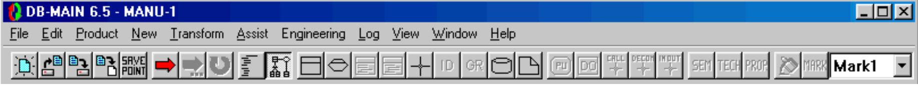


Figure 1.4 - The complete Menu bar and the full *Tool bar*.

1. Double-clicking does not work here, for reasons that will be explained later.
2. This window can also show all the activities that have been carried out to build these products. In other words, the Project window can show, if requested to, the history of the project. We will ignore this feature in the following lessons.

1.4 Defining a new schema

We create a new schema in which we will draw the conceptual structures of the database. Through the command **Product / New product** the *Schema box* appears and asks us the name (`Manufacturing`), the short name (`Manu`) and the version of the schema. This schema will include the conceptual description of our database in project, so that `Conceptual` should be a clear version name that suggests the objective of the schema.

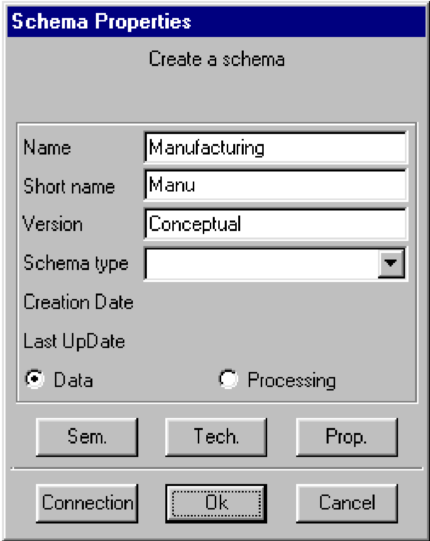


Figure 1.5 - Creating a new schema.

We ignore the other properties and we validate the operation by clicking on the `OK` button.

Two things happen. First, a new icon with the name `Manufacturing/Conceptual` appears in the Project window, indicating that the project comprises a new document, or product, which is a schema. Later on, double-clicking on such an icon will open its *Schema window*.

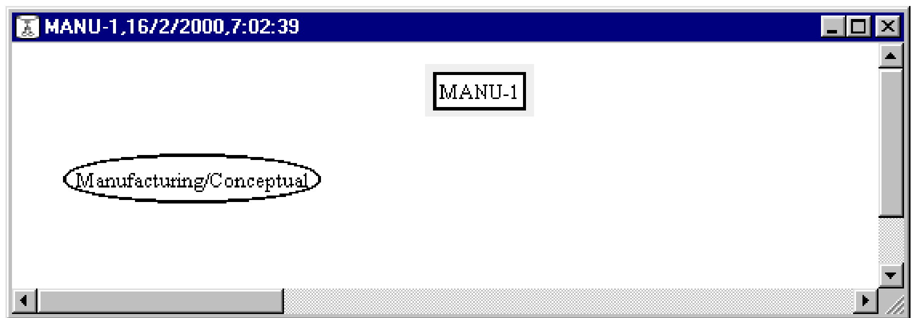


Figure 1.6 - The project window includes the new schema[3].

Secondly, a *Schema window* is opened, showing the same icon, but nothing else.

Figure 1.7 - The schema window is empty, except for the icon of the schema itself. This window is like a blank page on which we will draw the conceptual schema of the future database.

This icon represents the schema. Double-clicking on it opens its *Schema (property) box*. So far, this schema is empty. We will work in this window, so that it is a good idea to enlarge it.

3. In some rare situations (for instance, if you work on a DB-MAIN version already used by a professional who configured it differently) a small rectangle with the label New schema also appears in the Project windows. To get rid of it, check that the Project window display mode is *Graphical Dependency* (through **View** / **Graph. Dependency**). The other modes are quite nice as well, but probably a bit disturbing for an introductory lesson!

From now on, in order to simplify the illustrations used in this lesson, we will hide the schema object, except when needed.

Note. To free the workspace, especially when it is crammed with many windows, it is best to iconize (minimize) the Project window.■

1.5 Defining entity types COMPANY and PRODUCT

To enter the *create entity type* mode, we click on the button. That changes the cursor that now looks like a little rectangular box. We choose a point in the schema window, we put the cursor on it and we double-click. This lays an entity type at that point and opens the *Entity type box* that allows us to define a new entity type (Figure 1.8).

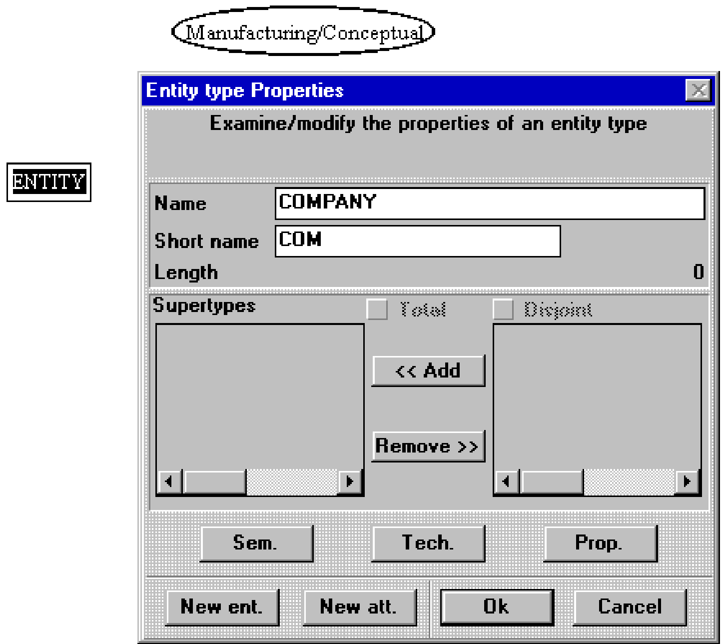


Figure 1.8 - The first entity type is defined.

We enter the name `COMPANY` and short name `COM`. We validate the operation by clicking on the `OK` button.

In the same way, we double-click at another point to define entity type `PRODUCT` with short name `PRO`. To quit the entry mode, we click on the *New Entity type* button again, or we press the `Escape` key.

Now, the schema window shows the newly defined entity types as two boxes. We move the boxes (by dragging them with the mouse) in the window in order to give the schema a nice layout (Figure 1.9)

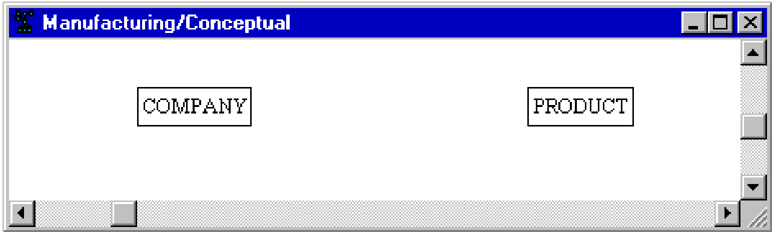


Figure 1.9 - So far, the current schema is made up of two entity types.

1.6 Entering entity type attributes

To specify that some specific information items are associated with the entities of each type, we will define the *attributes* of these entity types. We open the property box of entity type `COMPANY` by double-clicking on its name in the schema window, then we click on the `New att.` button. The *Attribute box* invites us to define the first attribute (Figure 1.10). We give it the name `Com-ID`, the type `char`(acter) and the length `15`. This attribute represents the company identifier, and is considered as a string of 15 characters. For now, we can ignore the other properties.

There are other attributes that we want to associate with `COMPANY`. Therefore, we click on button `Next att`(ribute), which validates the current definition, and which calls the Attribute box again (since this button is the active one, just pressing the `Enter` key will do it). We define successively attributes `Com-Name` (char 25), `Com-Address` (char 50) and `Com-Revenue` (numeric 12). The last attribute will be validated by clicking on the `Ok` button instead to stop the entry process.

Figure 1.10 - The first attribute of COMPANY is defined. The next attributes will be defined by pressing the `Next att.` button, or more simply by pressing the `Enter` key.

In the same way, we define attributes `Pro-ID` (char 8) and `Pro-Name` (char 25) of entity type `PRODUCT`.

The schema window now looks like Figure 1.11.

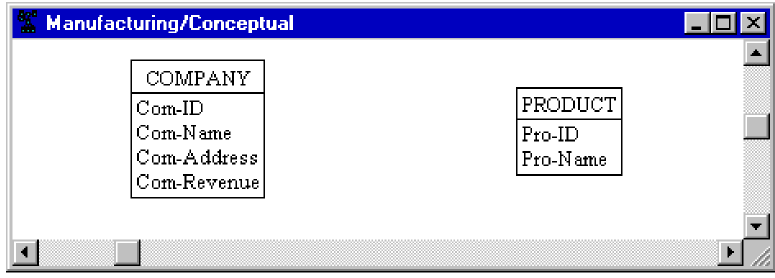


Figure 1.11 - The entity types have been given specific attributes.

1.7 Entering relationship type MANUFACTURES

Now we want to represent the fact that *companies manufacture products*. This can be done by drawing a relationship type (or *rel-type* for short) between these entity types.

We enter the *New rel-type* mode by clicking on the button in the Tool bar[4]. The cursor takes a cross-hair shape, so that we can draw a line from COMPANY to PRODUCT in the schema window (Figure 1.12).

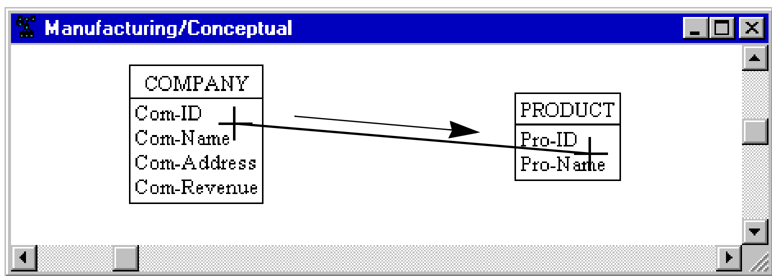


Figure 1.12 - A line is drawn between the boxes of the entity type we want to connect.

A link appears between both rectangles with a hexagon on it. Normally, the default name R is selected (white on black). If it is not, we click on it. We press the Enter key to open the *Rel-type box* (or we double-click on name R) We enter the correct name manufactures, then we validate through the Ok button (Figure 1.13).

We quit the entry mode just like we did for the entity types by pressing the Escape key or by clicking on the button again (or on any another entry button).

Each end of the rel-type is called a **role**. Each role is taken by an entity type and is given a *cardinality constraint*, that appears as a pair of symbols, such as 0-N and 1-1.

The 0-N cardinality specifies that any COMPANY entity will appear in at least 0 and at most N (standing for *infinity*) manufactures relationships.

---

4. or button in Version 3.

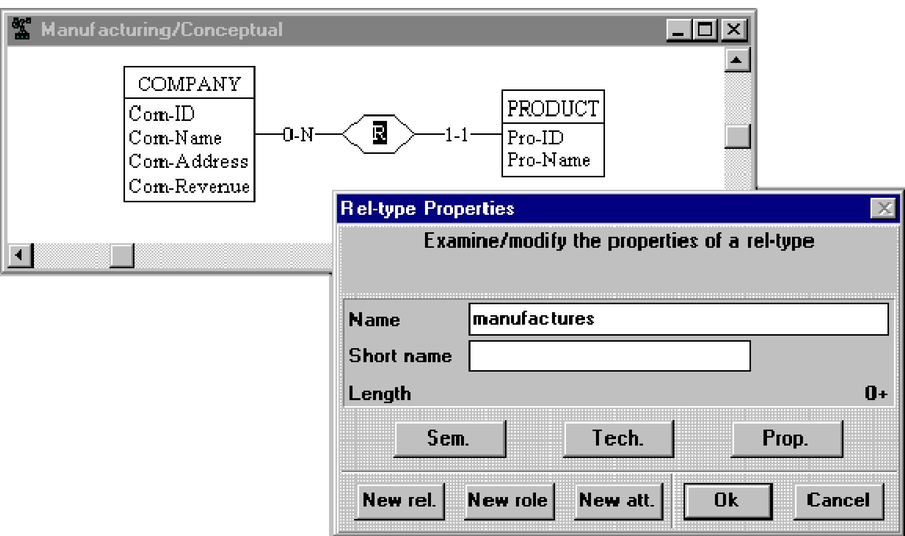


Figure 1.13 - A relationship type links the entity types. It will be given the name *manufactures*.

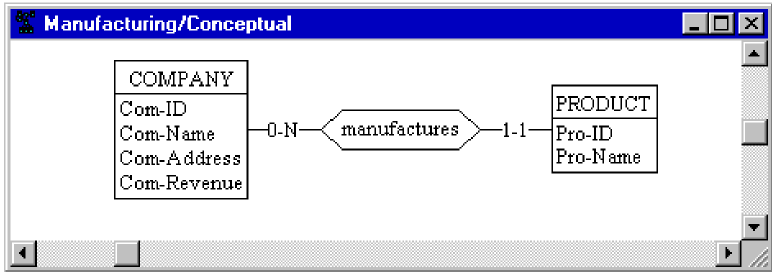


Figure 1.14 - Now the schema explicitly tells that *companies manufacture products*.

We will study later the concept of cardinality in greater detail. For now, we understand the `0-N` cardinality as "*a company manufactures an arbitrary number (i.e,. from 0 to N) of products*". Similarly, the schema shows that a `PRODUCT` entity will appear in exactly one (i.e., from 1 to 1) `manufactures` relationship.

The cardinality can be changed by double-clicking on the role, i.e., on its cardinality symbol. This will be examined in detail in another lesson.

1.8 Defining entity type identifiers

Normally, the entities of the same class, for instance all the companies, have a special property that allows us to designate each of them. This property is called an **identifier** of the entity type. Usually, it is a name, a code, a reference or anything else that makes the entities unique in their class.

For instance, we want to tell that Com-ID is the unique code of companies. We select this attribute by clicking on its name (which appears white on black) than we click on the Identifier button [ID] on the Tool bar.

In the same way, we define PRO-ID as the identifier of entity type PRODUCT. The schema can now be considered as complete (Figure 1.15).

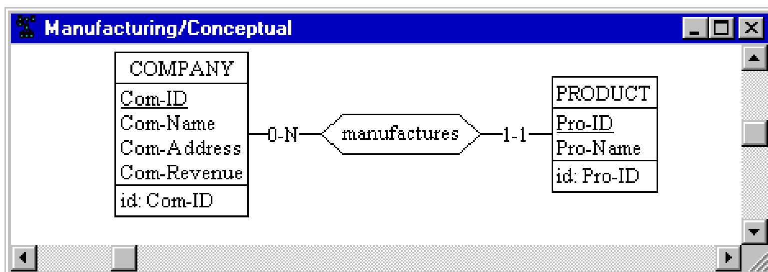


Figure 1.15 - An identifier has been associated with each entity type.

Note that the identifier is graphically mentioned twice (assuming the novice analyst has not noticed the fact!): first through the id clause that appears at the bottom of the entity type box, and secondly by the underlining of the component attribute. This latter way will be used when the identifier comprises attributes only.

1.9 Documenting the schema

You have probably observed that most boxes that define the properties of an object have a special button named Sem. Clicking on the Sem button opens a

small text window in which we are allowed to enter a free text that describes the meaning of the current object, i.e., its *semantics*.

Let us double-click on the COMPANY entity type (another way: select COMPANY, then press the Enter key). We get the Entity type property box of COMPANY. We click on the Sem button, and we enter a text that defines what a company is (Figure 1.16).

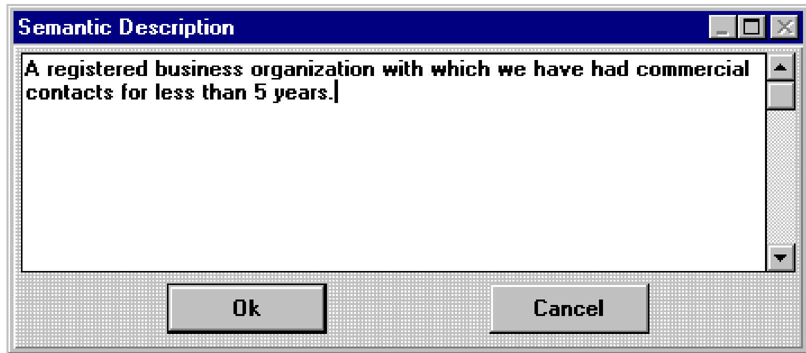


Figure 1.16 - The *Semantic description* text window of an object.

The text can be as long as needed (with a 32 Kb limit however). It can be cut, copied and pasted from/to any other program in the usual way (ctrl-X, ctrl-C, ctrl-V).

In the same way, we can enter a description for PRODUCT and manufactures, for each of the attributes, for each role, for each identifier and even for the schema and the project themselves.

Note. There is a similar button SEM on the *Standard tools* bar which has the same effect: select any object in the current schema, then click on this button to open the Semantic description window of the object.■

1.10 Producing a SQL[5] database

There are several ways in which this conceptual schema can be translated into table and column structures. For now, we have no special requirements as far as performance, or any other consideration, are concerned. We will be happy with an unsophisticated translation of this schema into SQL commands.

This translation can be done in a straightforward way through the command **Transform / Quick SQL**. DB-MAIN simply asks you, with the standard file dialog box, in which file you want the SQL program to be stored. By default, the file will be named `manu-1.ddl`, following the name of the project (Figure 1.17).

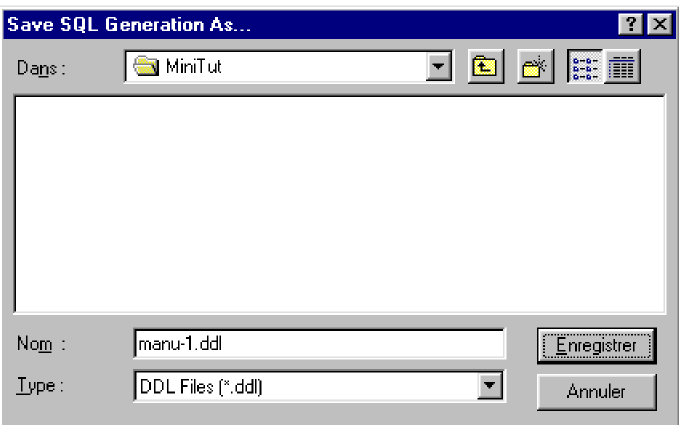


Figure 1.17 - The SQL program that is being generated from the conceptual schema will be saved as `manu-1.DDL` file.

Now, we go back to the Project window. We observe that a new product has been made available. The slightly different icon shape indicates that this new document is a text file called `manu-1.ddl`. Obviously, this is the SQL program we just generated in the last step.

We can examine the contents of this text file by double-clicking on its icon. A new text window opens, showing the SQL code implementing the conceptual schema. It should read like in Figure 1.19.

---

5. SQL must be read SEQUEL.

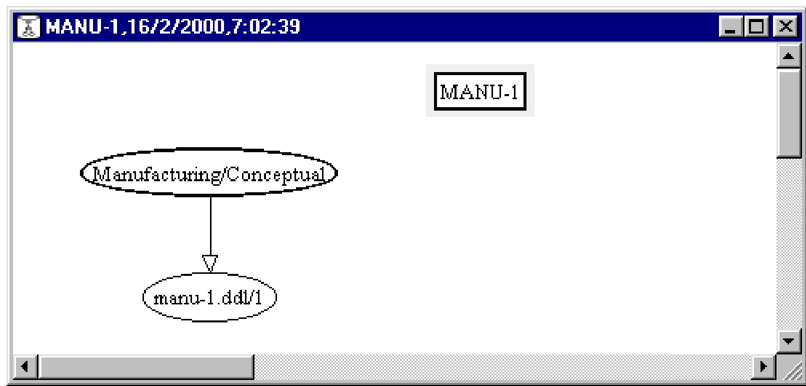


Figure 1.18 - Now, the project window includes two products, namely the conceptual schema and the SQL program that derives from it.

```
create database Manufacturing;

create table COMPANY (
       Com-ID char(15) not null,
       Com-Name char(25) not null,
       Com-Address char(50) not null,
       Com-Revenue numeric(12) not null,
       primary key (Com-ID));

create table PRODUCT (
       Pro-ID char(8) not null,
       Pro-Name char(25) not null,
       Com-ID char(15) not null,
       primary key (Pro-ID));

alter table PRODUCT add constraint FKmanufactures
            foreign key (Com-ID) references COMPANY;

create unique index IDCOMPANY on COMPANY (Com-ID);
create unique index IDPRODUCT on PRODUCT (Pro-ID);
create index FKmanufactures on PRODUCT (Com-ID);
```

Figure 1.19 - The contents of the `manu-1.ddl` text file can be examined by double-clicking on its icon in the project window.

To be quite precise, this SQL program will not necessarily be executable on all machines, and would probably need some syntactic adjustements. For instance, dashes ("-") are not allowed by most SQL DBMS, and should be replaced by, say, underscores ("_"). We will see later how this kind of problem can be addressed in a systematic way.

In addition, the set of indexes may not be the most efficient one, and would need some refinement. Such decisions relate to physical design, an activity that obviously is far beyond the scope of this first lesson!

1.11 Saving the project

As is natural after working such a long time, we carefully save our work through command **File / Save project** (or button) or command **File / Save project as** (or button) in order to make it available for further use.

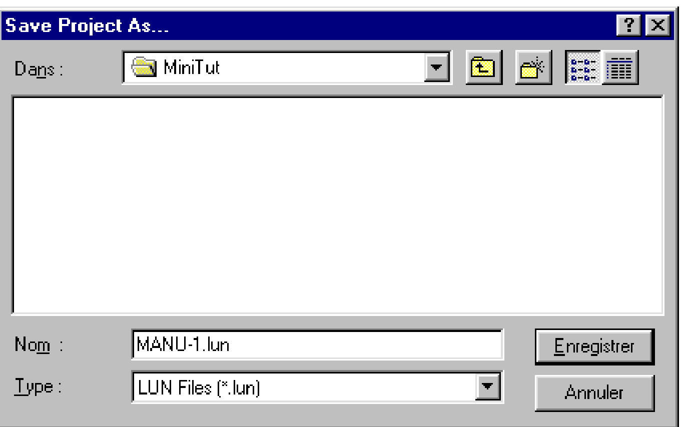


Figure 1.20 - The whole project is saved on disk.

By default, the project is saved as file `manu-1.lun`. We validate the operation through the button `OK`.

The `*.lun` extension is typical to the saved DB-MAIN projects, so do not use them for other files.

1.12 Quitting DB-MAIN

It is now time to exit from the DB-MAIN tool by command **File / Exit**.

We have built our first SQL database, and we are now able to build other simple SQL databases just by applying the basics that have been presented in this lesson.

Summary of Lesson 1

- In this first lesson, we have studied some important concepts:
  - the concept of CASE tools
  - projects and schemas
  - entity types, relationship types, attributes and identifiers
  - conceptual schemas
  - SQL expression of a conceptual schema

- We have also learnt to:
  - run the DB-MAIN CASE tool
  - create a new project: **File / New project**
  - create a new schema: **Product / New schema**
  - define an entity type: **New / Entity type**
  - define an attribute: **New / Attribute**
  - define a relationship type: **New / Rel-type**
  - define an identifier: **New / Group**
  - add a semantic description:
  - save the current project: **File / Save as**
  - save the current project: **File / Save**
  - produce SQL code: **Quick DB / SQL** or **Transform / Quick SQL**
  - exit from DB-MAIN: **File / Exit**

- We have produced two types of files:
  - saved projects (`*.lun`)
  - executable code such as SQL (`*.ddl`).

Exercises for Lesson 1

Define a project, a conceptual schema and generate an SQL database creation program for each of the situations described below.

1.1 The small database we developed in this lesson was based on the hypothesis that a product is manufactured by one company only (cardinality 1-1). Now, consider that *a product can be produced by any number of companies* (i.e., by 0, 1, 2, or more companies). Change the schema accordingly. Don't save this project.

1.2 Customers buy products in such a way that each customer can buy any number of products and each product can be bought by an arbitrary number of customers. Imagine some natural attributes for the entity types. Call this project `SALES1` and save it.

1.3 Students belong to classes: each student belongs to exactly one class (no less, no more), while a class comprises any number of students. Each student can be registered in any number of courses while any number of students can be registered for a given course. Imagine some natural attributes for the entity types. Call this project `STUDENT1` and save it.

1.4 Complete the `MANU-1` project by considering countries to which products are exported.

Don't save the modified project (we will make use of the original version in further lessons), unless you give it another name.

Lesson 2

A closer look at schemas

Objective

This is an easy and relaxing lesson (just playing with existing schemas!). It presents some useful schema display formats and the way to use them. In this lesson, we also study how to manipulate graphical and textual objects, how to change their apparent size, how to navigate through a schema and to generate reports.

Preliminary checking

In this lesson, we will use the project `MANU-1` (file `manu-1.lun`) that has been created in Lesson 1, and the `LIBRARY` project (or its French equivalent `BIBLIO`) that comes with the DB-MAIN software.

2.1 Starting Lesson 2

Let us start DB-MAIN and open the project `MANU-1` through the command **File** / **Open project** or by clicking on the button . When the project is opened, we double-click on the icon of the `Manufacturing/Conceptual` schema to display its contents.

For this lesson, we will need some new functions that are offered by the menu, but that are available on a new tool palette as well. We display this new palette through **Windows** / **Graphical tools** (Figure 2.1). These tools can be placed anywhere on the screen, for instance under the *Standard tool bar*.

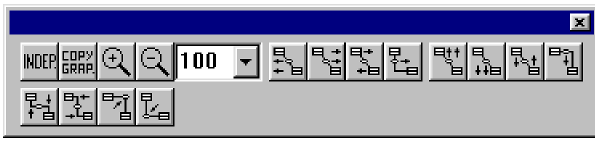


Figure 2.1 - The graphical tool bar. It can be resized according to your taste.

2.2 On including database schemas into a document

In the first lesson, several figures include a schema, showing the step-by-step construction of the conceptual description of our database. As everybody should have observed, these schemas have been obtained from screen copies. This technique provides nice looking results, but is rather painful (the screen shots have to be processed with an image processing software) and yields huge documents.

The DB-MAIN tool includes a function that copies selected schema objects onto the clipboard in a more concise format (as vector-based objects). So, select all the objects of the schema, then call the **Edit** / **Copy graphic** menu item or click on the button in the *Graphical tools* bar. Then, open a Word or

Powerpoint document, and paste the clipboard contents (use *Paste* or *Paste Special* according to the software).

The schema objects appear in the text document as in Figure 2.2 (bottom). The result can be modified as any vector-based graphical object[1]. From now on, we will use this technique to include schema fragments in this lesson and in the next ones.

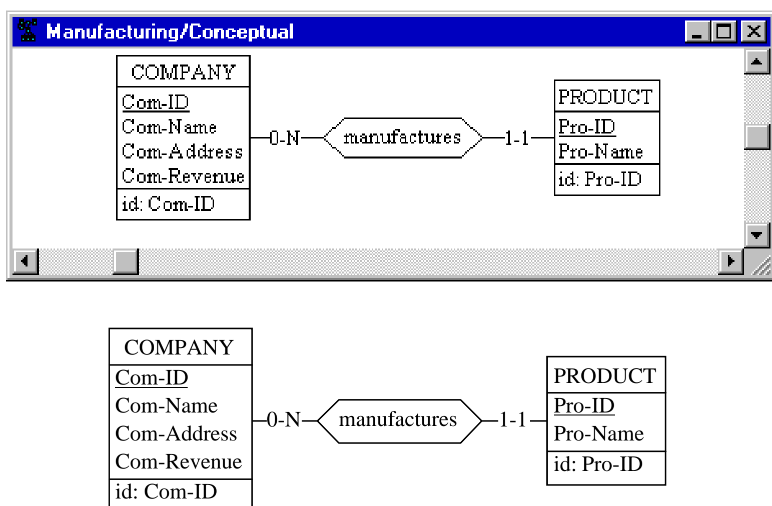


Figure 2.2 - Bitmap (top) and vector-based (bottom) schemas as they appear in a text document.

2.3 Graphical views of a schema

In Lesson 1, the schema was represented in a Schema window through graphical objects. There are several other ways to display this schema. They can be classified into *graphical views* and *textual views*. This section is devoted to graphical views.

1. In some products, such as MS-Word or FrameMaker, the labels may appear to be too long or too short for the rectangles in which they are enclosed after the schema has been redimensioned. This is due to the way Windows redimensions a graphical object: continuously for geometrical components and point by point for texts. In this case, just expand or stretch the schema frame **horizontally** until the texts correctly fit in their boxes.

Let us first examine a new way of presenting large schemas, namely the *compact view*. It can be obtained through the **View** / **Graph. compact** command. The attributes and identifiers are hidden in such a way that only the schema *skeleton* appears (Figure 2.3).

COMPANY —0-N— manufactures —1-1— PRODUCT

Figure 2.3 - The *compact graphical* view of the MANU-1/Conceptual schema.

Now, we go back to the standard graphical view through **View** / **Graph. standard**, to get the view we have used so far (Figure 2.4). Since this view is the most useful, it has been given a special button on the Standard tools bar: .

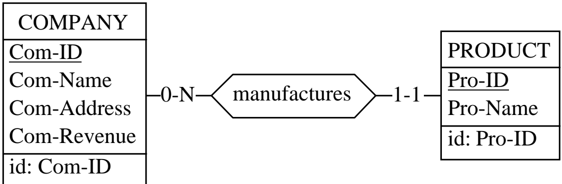


Figure 2.4 - The *standard graphical* view of the MANU-1/Conceptual schema.

Starting from this standard view, we can derive some simplified forms by using the graphical settings panel (**View** / **Graphical settings**) (Figure 2.5).

The buttons of the *Show Objects* block of this panel can be unchecked, which hides the attributes, or the identifiers (called *groups* in the panel), or both (Figure 2.6). You can also show the attribute types if needed.

Graphical variants exist to represent entity types and rel-types. For instance, we can choose to draw entity type and/or rel-type boxes with round corners instead of square ones by selected *rounded* shape in the Graphical settings panel (Figure 2.7). These settings are valid for the current schema. They can be useful to distinguish different levels of schemas.

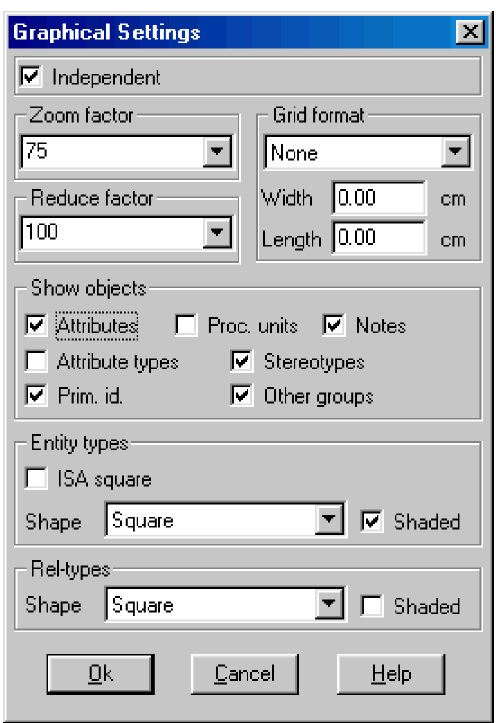


Figure 2.5 - The *Graphical settings* panel.

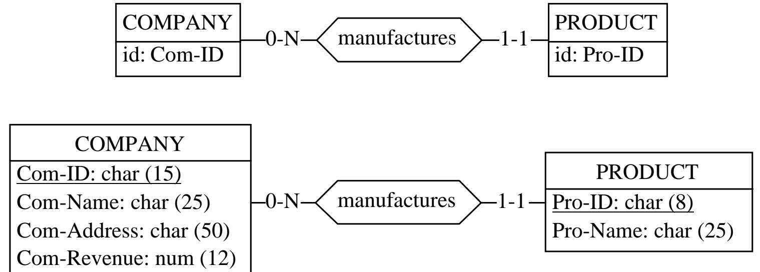


Figure 2.6 - The Standard view *without Attributes* (top) and *without Groups (i.e., without identifiers)* but *with attribute types* (bottom).

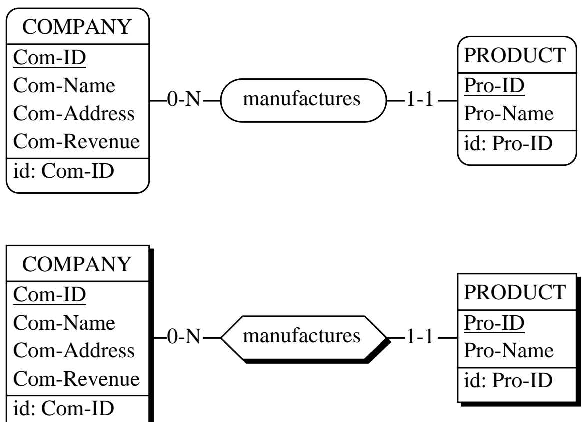


Figure 2.7 - *Round-corner shape* and *shaded boxes* as alternate graphical representations.

A last trick before leaving the graphical views of a schema: *how to retrieve a* ***selected object*** *in a schema*. Let us suppose that the (small) schema window shows a fragment of a (large) schema. Let us also suppose that an object is selected, somewhere in the schema, but not shown in the window. How to move the window in such a way that the selected object is at the center of this window? Nothing can be simpler: just press the **tab** key.

What if there is more than one selected object? The **tab** key brings the *next selected object* in the window.

2.4 Textual views of a schema

The contents of a schema can be presented as a pure text as well. In this mode, four formats are available.

The simplest one is the *compact view*. It shows a mere list of the names of the entity types followed by that of the relationship types (Figure 2.8).

```
                    Schema Manufacturing/Conceptual

COMPANY
PRODUCT

manufactures
```

Figure 2.8 - The **Text compact** view of a schema

This list is a sort of dictionary. It can be obtained through the command **View / Text Compact**.

The compact view does not display the detail of a schema and can be used as a quick index to locate an object in a large schema.

For a more detailed textual view, try the *Standard view*. It can be obtained through the command **View / Text Standard**, and presents the current schema as in Figure 2.9. Since it is frequently used, it can also be obtained through a specific button on the Standard tools bar: .

```
                    Schema Manufacturing/Conceptual

COMPANY
     Com-ID
     Com-Name
     Com-Address
     Com-Revenue
     id: Com-ID

PRODUCT
     Pro-ID
     Pro-Name
     id: Pro-ID


manufactures(
     [1-1]:: PRODUCT
     [0-N]: COMPANY)
```

Figure 2.9 - The **Text standard** view of a schema

The *extended view* is an even more complete presentation. In addition to the information of the standard view, the *extended view* shows, among others, the short names, the type and length of the attributes and the roles in which each entity type appears. The symbol `[S]` indicates that a semantic description has been associated to the object.

This view is obtained through the command **View** / **Text extended**, and appears as in Figure 2.10.

```
            Schema Manufacturing/Connceptual / Manu [S]

COMPANY / COM [S]
     Com-ID  char (15) [S]
     Com-Name  char (25) [S]
     Com-Address  char (50) [S]
     Com-Revenue  numeric (12) [S]
     id: Com-ID
     role: [0-N] in manufactures

PRODUCT / PRO [S]
     Pro-ID  char (8) [S]
     Pro-Name  char (25) [S]
     id: Pro-ID
     role: [1-1] in manufactures


namufactures [S] (
     [1-1]: PRODUCT
     [0-N]: COMPANY)
```

Figure 2.10 - The **Text extended** view of a schema. The directed arcs show the possible jumps through the hyperlinks activated by a right-button click.

Note that the *role lines* that appear both in the entity type and rel-type paragraphs makes it possible to navigate through the whole schema by jumping from an entity type to the relationship types in which it appears, and conversely:

- to jump from an entity type to one of its relationship types: click on the line of the role in the entity type paragraph *with the **right** button* of the mouse.
- to jump from a relationship type to one of its entity types: click on the line of the role in the rel-type paragraph *with the **right** button* of the mouse.

These *hyperlink* functions are very handy for large schemas. More on schema navigation later in this lesson.

The last format is the *sorted view*, which presents an unstructured sorted list of *all the names* that appear in the schema, together with their type and origin. This view is particularly important for large and complex schemas, specially in *reverse engineering* activities[2]. It can be used too when checking names in conceptual analysis. In addition, it is the easiest way to retrieve an object when only its name is known.

The sorted view can be obtained through the command **View** / **Text sorted**, and appears as in Figure 2.11.

```
                Schema Manufacturing/Conceptual

Com-Address     Att. of COMPANY
Com-ID          Att. of COMPANY
Com-Name        Att. of COMPANY
Com-Revenue     Att. of COMPANY
COMPANY         Entity type
manufactures    Rel-type
Pro-ID          Att. of PRODUCT
Pro-Name        Att. of PRODUCT
PRODUCT         Entity type
```

Figure 2.11 - The **Text sorted** view of a schema

Two important properties

- Objects that are selected (highlighted) in a view still are selected in any other view in which they appear. For instance, an attribute with a particular name can be retrieved in a schema by using the *text sorted view*. Now, choosing the *standard graphical view* allows us to examine this attribute in its context.

2. *Reverse engineering* can briefly be described as the converse of what we did in the first lesson, that is *recovering the conceptual schema of an existing database*. It involves complex techniques and tools that are described in other documents but that will be ignored in this tutorial.

- Building a schema, or examining, deleting and modifying its components, can be performed whatever the view in which this schema is displayed. For instance, double-clicking on the line of an object in a text view opens the same property box as in a graphical view.

2.5 Manipulating the graphical components of a schema

Now, let us go to a graphical view of the schema. The position of the objects of this schema can be changed by *selecting and dragging* them in the usual way. Several objects can be selected (or deselected) by pressing the `shift` key when selecting, or by drawing a selection rectangle with the mouse, and moved simultaneously.

Moving objects

Moving objects in their window obeys the general Windows rules:

- selected objects are moved by dragging them in the window space;
- selected objects are moved by pressing the cursor keys (← ↑ → ↓);
- small-step moves are obtained by pressing the cursor keys while pressing the Ctrl key;
- using the scroll bars moves the window in the four directions.

The ***Move mode*** designates the way DB-MAIN reacts when an object is moved on the screen: does it move the object only (*independent* mode), or does it reposition the connected objects as well (*dependent* mode)? This mode can be set either in the *Graphical settings panel* (`Independent` button) or through the `INDEP.` button on the *Graphical tools* bar: INDEP.

In the *Dependent mode*, the graph is adjusted as follows (Figure 2.12 left):

- when an entity type is moved, its relationships types and their roles are moved proportionally and redrawn;
- when a relationship type is moved, its roles are moved too,
- when a role is moved, nothing else is redrawn.

In the *Independent mode*, the graph is adjusted as follows (Figure 2.12 right):

- when an object (entity type, relationship type, role) is moved, nothing else is redrawn, except the arcs that link it to the other objects.

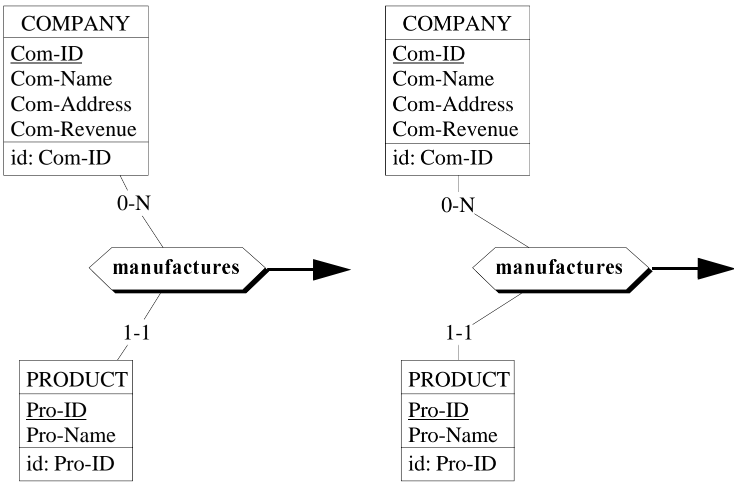


Figure 2.12 - Moving rel-type `manufactures` in the *dependent mode* (left) and in the *independent mode* (right).

Aligning objects

After a while, a schema may look like spaghetti, and we might want to put some order among its components. A first nice feature is the rel-type ***Align*** action which allows us to align a role or a relationship type according to its connected objects. We can get this effect by clicking on the object (role or rel-type) with the *right button* of the mouse (Figure 2.13).

To align a larger set of objects, we will make use of the **View** / **Alignment** command, that provides us with eight operators, four for vertically aligning the objects and four for horizontal alignment. They are also available on the *Graphical tools* bar (Figure 2.14).

In the **horizontal** dimension, we can align objects on their left side, on their right side, we can center them and we can distribute them horizontally at equal distance.

In the **vertical** dimension, we can align objects on their top side, on their bottom side, we can center them and we can distribute them vertically at equal distances.

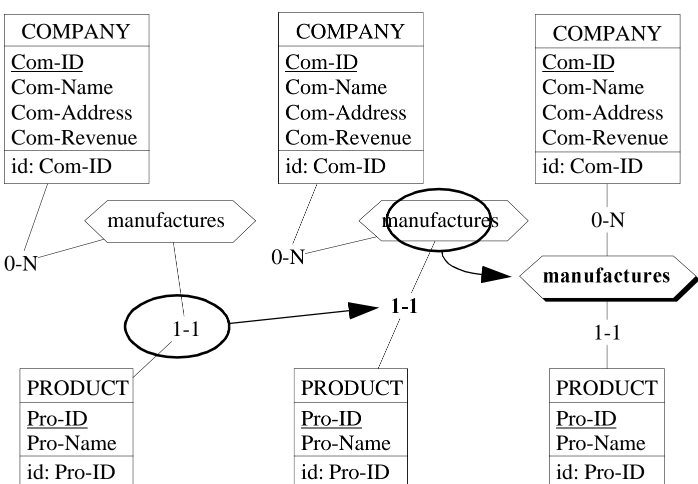
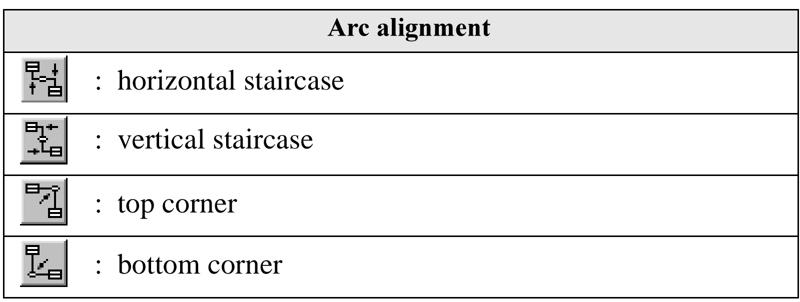


Figure 2.13 - Aligning roles (center) and relationship types (right) by clicking with the **right** button of the mouse.

Horizontal object moves	
[icon]	: align to left
[icon]	: align to right
[icon]	: center horizontally between left and right
[icon]	: distribute evenly between left and right
Vertical object moves	
[icon]	: align to top
[icon]	: align to bottom
[icon]	: center between top and bottom
[icon]	: distribute evenly between top and bottom

Figure 2.14 - The eight object alignment operators.



Arc alignment	
	: horizontal staircase
	: vertical staircase
	: top corner
	: bottom corner

Figure 2.15 - The four arc alignment operators.

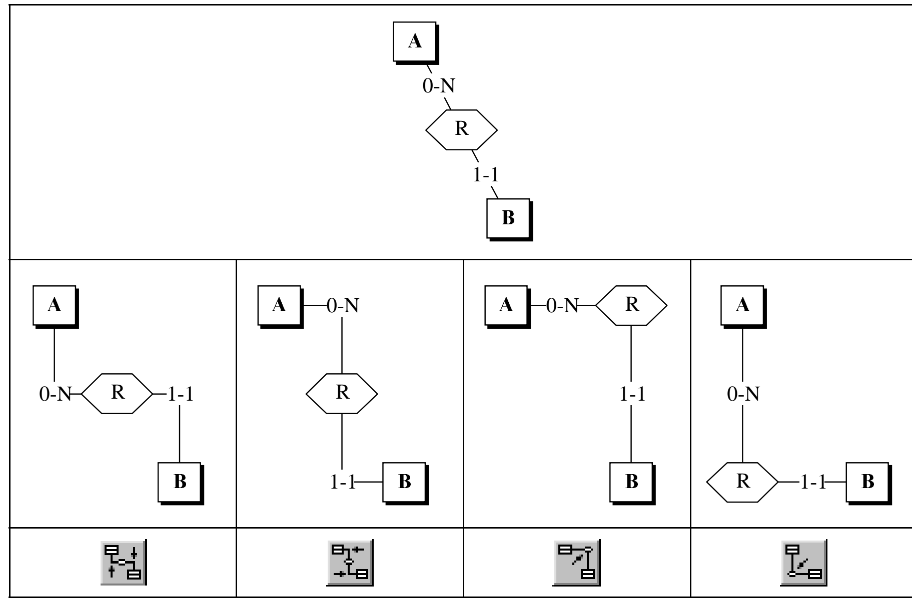


Figure 2.16 - How to draw a source rel-type (top) with *staircase* style with just a mouse click.

Two comments:

1. *Horizontal* means that the objects are moved *horizontally* to reach their final position (the same for the *vertical* direction).
2. When the objects are distributed evenly, the distance is evaluated between the edges of the objects, not between their centers. This provides a natural positionning of roles and rel-types between their entity types.

The last four alignment operators (Figure 2.15) are dedicated to users who are found of *staircase* rel-types. Since an image is worth one thousand words, we suggest you had a look at Figure 2.16.

The best way to get acquainted with these operations is to play with a disaligned schema such as that of Figure 2.17, which is available in project `Manu-3.lun`, schema `Alignment`.

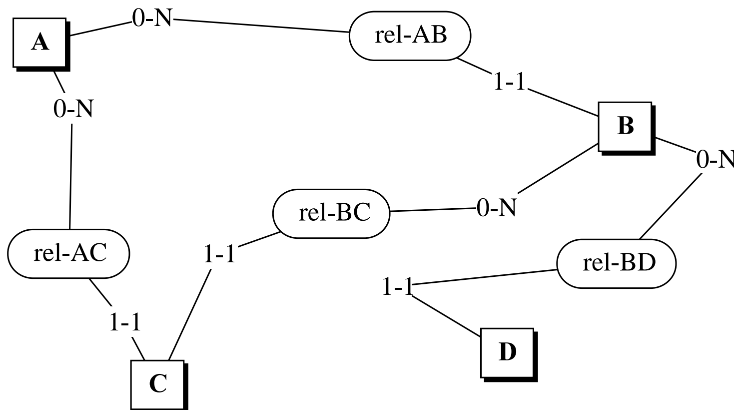


Figure 2.17 - This schema obviously suffers from a severe disalignment disease. Cure it.

Zooming in and out

For large schemas, a ***zooming function*** is available to help fit a larger or a smaller portion of the schema in the Schema window (zoom out), or to examine tiny details (zoom in). This function is available in the *Graphical settings panel* (Figure 2.5) and in the *Graphical tools* bar (Figure 2.1) through the following buttons:

expands the schema representation by 10%;

shrinks the schema representation by 10%;

sets the zoom factor by specifying its exact value; the `fit` value adjusts the zoom factor so that the schema fits in the schema window.

Using a larger schema

To get a better feeling of the usefulness of the various views, we switch to another project. We close the current one (command **File** / **Close project**), and we open the `LIBRARY` project (command **File** / **Open project**) and its schema. Now we experiment with each view, and try to figure out the meaning of the components of this schema, which obviously describes the management of a scientific library. Its contents include many more modeling characteristics that will be discussed later.

Last observations

We observe that:

- switching from a view to another one is immediate, and can be asked for at any time;
- the operations of the tool are independent of the view through which they are executed;
- an object that is selected (highlighted) in a view still is selected when we switch to another view;
- if several schemas of a project are opened (more on this later on), they can be displayed in different views.

2.6 Navigation through textual views

When a schema is small, it spans one or two screens only. Retrieving an object in such a schema needs no special skill nor any special tool. The problem is less trivial when the schema is larger, and is several dozens of screens large (large schemas can include thousands of entity types and rel-types): browsing through such a schema can be time consuming and does not garantee that the objects we are looking for will be found quickly, if ever.

Retrieving a specific object can often be made easier by working first on the **Text compact** and **Text sorted** views, using them as some kind of dictionaries, then switching to the standard graphical or text views when the object of interest has been found.

Another useful tool for object retrieval in context is the navigation feature of DB-MAIN. To illustrate them, we need a larger schema, such as LIBRARY. We display it in the **Text extended view**, and we reduce the Schema window a little bit to simulate a *large schema in a too small window*.

Let us experiment the navigation capabilities of DB-MAIN. Unless told otherwise, the following manipulations are valid for the **Text standard** and **Text extended** views.

- We select the COPY entity type by clicking on its name; we observe that each line in which the name COPY appears (i.e., each instance of COPY) is tagged with symbols ">>"; such is the case for each role in which COPY appears;
- If we press the TAB key; the next tagged instance of COPY appears in the center line of the Schema window; this allows the cursor to jump to each of the relationship types in which COPY takes part;
- We click with the *right button* on a line describing a role in which COPY appears, in a rel-type paragraph; the COPY entity type is then selected; the right button acts as a *go home* button;

In the **Text extended** view, we click with the *right button* of the mouse on a role in which COPY appears, in its entity type paragraph, then click; the relationship type of the role is then selected.

In Figure 2.18, the navigation rules are shown on the small project `Manu-1`.

2.7 Reordering attributes and roles

Though the order in which attributes (and roles) appear in the textual and graphical views does not matter in most situations, you may want to change this order.

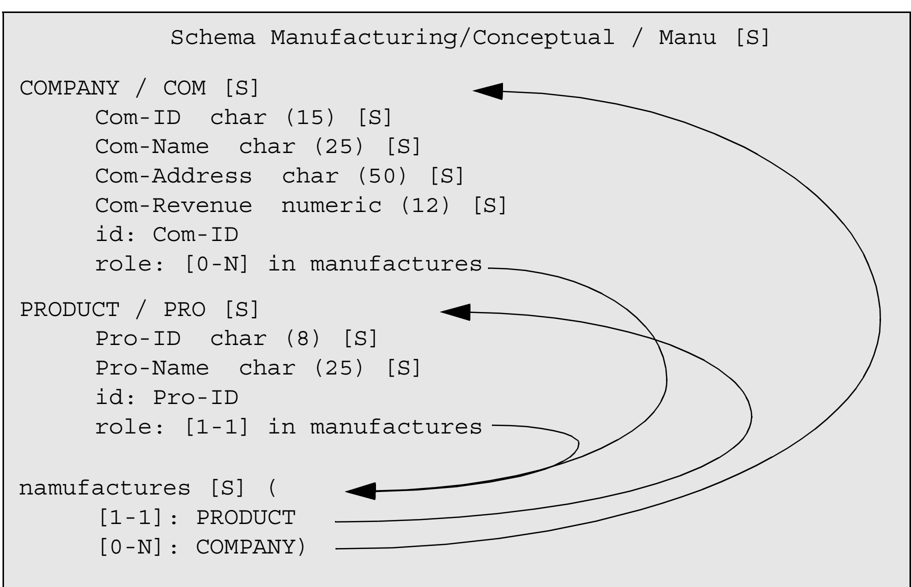


Figure 2.18 - Navigating in the **Text extended** view of a schema of project `Manu-1` with the *right button* of the mouse.

To **change the position of an attribute** (graphical and text views), select it, then

- press the Alt + ↑ keys[3] to move it one position up,
- press the Alt + ↓ keys to move it one position down (Figure 2.19).

To **change the position of a role** (text views), select it, then

- press the Alt + ↑ keys to move it one position up,
- press the Alt + ↓ keys to move it one position down.

The keys must be pressed simultaneously, not sequentially.

There are other ways to reorganize the attributes of an entity type, but they require more sophisticated functions (namely schema transformations) that will be studied later.

3. The keys must be pressed simultaneously, not sequentially.

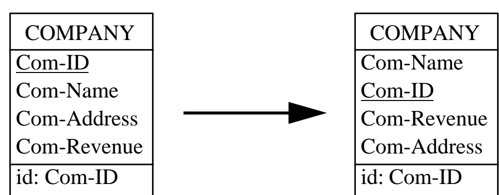


Figure 2.19 - Changing the order of the attributes with Alt + ↓↑.

2.8 Generating reports

A decent CASE tool must produce external documents that can be printed on paper. This one does it too. Several kind of reports can be of interest, ranging from simple object lists to sophisticated documents including a table of contents, an index and footnotes. Though DB-MAIN can produce such documents, we will show how to generate simple outputs.

1. First, we visualize our schema in any textual view (for instance with button).
2. Then we execute the command **File** / **Report** / **Textual view**. We get the panel of Figure 2.20.

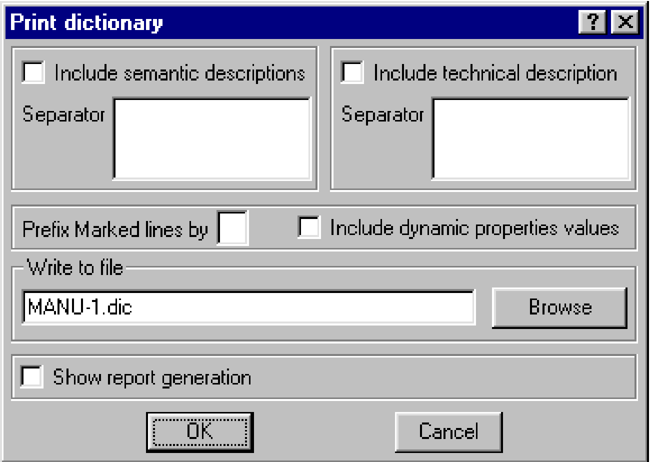
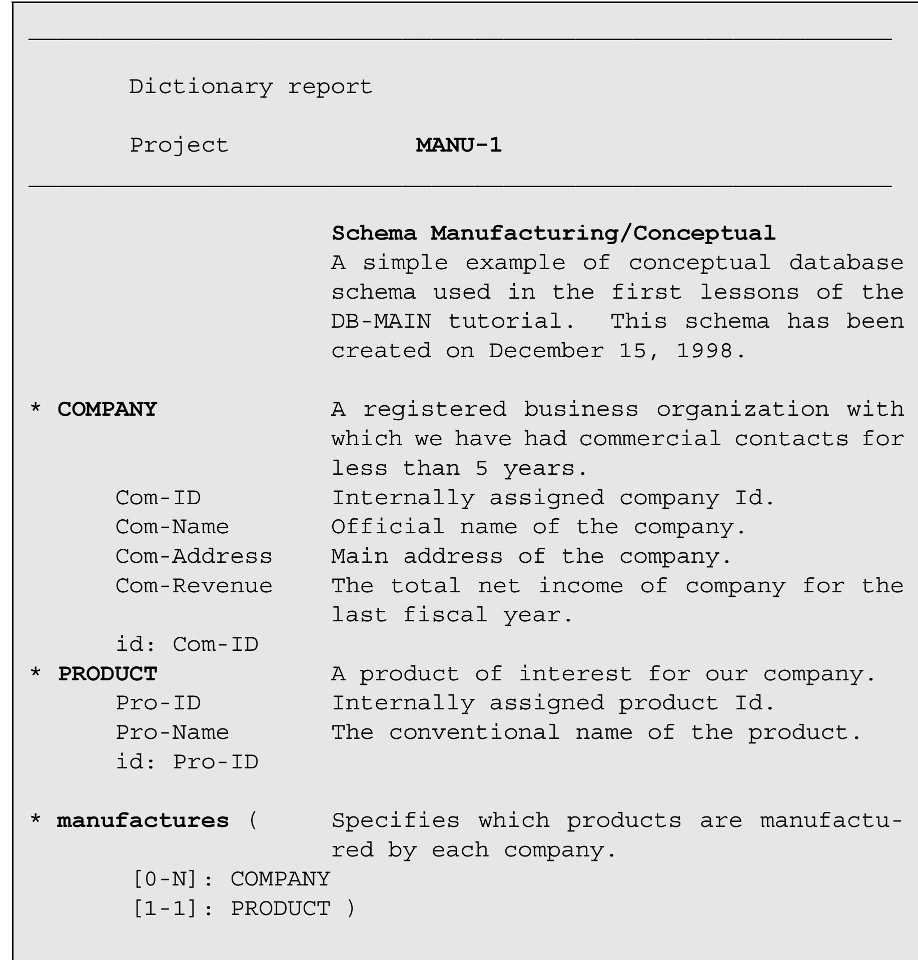


Figure 2.20 - Generating a simple text report.

3. We accept (or change) the default name (`MANU-1.dic`) of the output file in which the text will be stored. We ask for the semantic description to be included (check button Include semantic description). We define the character string that will be included just before each semantic description (a *tab* control can be used to clearly separate it from the object description[4]).
4. We click on OK.



```
____________________________________________________________

       Dictionary report

       Project             MANU-1
____________________________________________________________

                     Schema Manufacturing/Conceptual
                     A simple example of conceptual database
                     schema used in the first lessons of the
                     DB-MAIN tutorial.  This schema has been
                     created on December 15, 1998.

* COMPANY            A registered business organization with
                     which we have had commercial contacts for
                     less than 5 years.
      Com-ID         Internally assigned company Id.
      Com-Name       Official name of the company.
      Com-Address    Main address of the company.
      Com-Revenue    The total net income of company for the
                     last fiscal year.
      id: Com-ID
* PRODUCT            A product of interest for our company.
      Pro-ID         Internally assigned product Id.
      Pro-Name       The conventional name of the product.
      id: Pro-ID

* manufactures (     Specifies which products are manufactu-
                     red by each company.
       [0-N]: COMPANY
       [1-1]: PRODUCT )
```

Figure 2.21 - A simple text report.

4. According to the Windows conventions, a **tab** control is entered as *Ctrl* + *Tab*.

The result is a plain ASCII file which can be, if needed, further formatted with any word processor, to produce something like the text of Figure 2.21.

For immediate needs, we can directly send the current schema to the printer, be it in graphical or textual view, through command **File** / **Print**. The printer can be chosen and configured through **File** / **Printer setup** as usual.

There are other ways to produce reports. Let us remember one of them: the *Copy graphic* function, that allows us to include fragments of schemas into standard texts (Section 2.2).

2.9 Copying objects

When building a schema, it can happen that several entity types have to be given similar attributes, or that the schema includes parts that are almost the same. Instead of entering the similar objects manually, it could be more convenient to copy the original fragment, then to modify the copy.

The procedure is as expected:

1. select the components to copy and put them on the clipboard (ctrl+C or **Edit** / **Copy**);
2. paste them in the schema (ctrl+V or **Edit** / **Paste**);
3. if the the pasted objects are attributes, first select an entity type, a rel-type or an attribute; the pasted objects will be inserted after this insertion point (Figure 2.22).

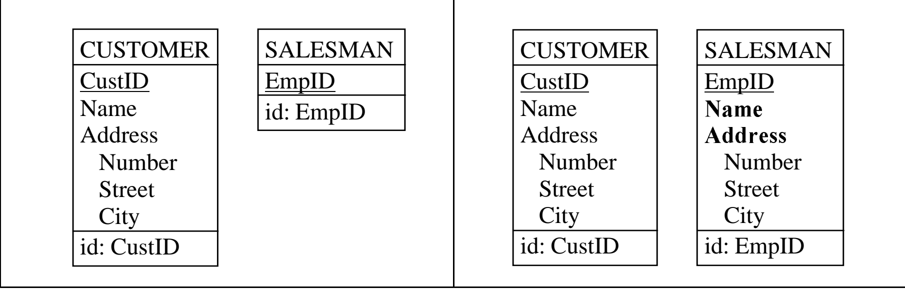


Figure 2.22 - It appears that `SALESMAN` must be given attributes similar to `Name` and `Address` of `CUSTOMER` (left). Select the latter, type ctrl+C, select `EmpID` of `SALESMAN` then type ctrl+V (right).

If needed, DB-MAIN makes the names of the pasted objects unique through the addition of a small suffix.

2.10 Pasting notes

To improve the understandability of the schema by adding free text information, notes can be asociated with an object or simply pasted on the schema, in the same way you *paste a post-it* on an object on in a document. To do so, we select the target object, we click on the note button ( ) then we click in the schema where you want the note to be pasted. The note can be opened by double-clicking on it and a text can be added (Figure 2.23, bottom). If no object is selected, the note is associated with the schema as a whole (Figure 2.23, top).

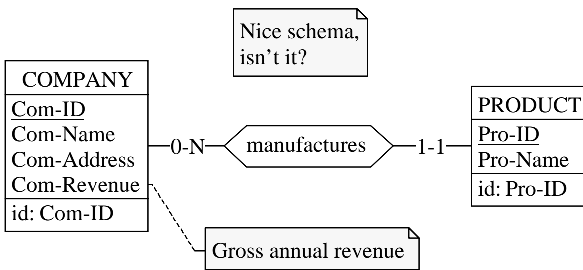


Figure 2.23 - Associating a note with an object or with the whole schema

2.11 Quitting the lesson

We will still use this project later on. Therefore, we save it with the name `manu-2.lun` and we quit DB-MAIN.

Summary of Lesson 2

- In this first lesson, we have studied some important concepts:
  - graphical views of a schema: compact, standard
  - text views of a schema: compact, standard, extended, sorted
  - navigation through the objects of a schema
  - graphical aspects of a schema (zoom and reduce)
  - text navigation through *role* links
  - reordering attributes and roles
  - simple reports

- We have also learnt:
  - to open an existing project:

    Project / Open project
  - to open an existing schema
  - to include fragments of a schema into a text:

    Edit / Copy graphic
  - to select a schema presentation format:

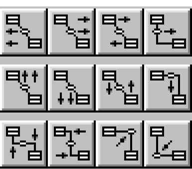
    View / Text compact
    View / Text standard

    View / Text extended
    View / Text sorted
    View / Graph. compact
    View / Graph. standard
  - to give graphical objects rounded corners and shades:

    View / Graphical settings
  - in a graphical view, to add associate a note with an object

    New / Note
  - in a text view, to navigate *from entity type to rel-type* and *from rel-type to entity type*: right button on the role line

- in a graphical view, to get the next selected object in the center of the schema window: tab key
- to move objects in the schema {← ↑ → ↓} and Ctrl + {← ↑ → ↓}
- to change the move mode of objects **View** / **Graphic**. **settings**
- to align rel-types and roles right button of the mouse
- to align a set of objects **View** / **Alignment**



- to zoom on a schema in and out **View** / **Graphic**. **settings**

- to reduce or expand a schema **View** / **Graphic**. **settings**
- to retrieve instances of an entity type in a text schema
  tagged lines *and* **tab** key
- to navigate between entity types and rel-types in a text schema
  right button of the mouse
- to change the order of attributes and roles in an entity type
  alt + ↑ ↓
- to copy selected objects elsewhere in the schema or in another schema of the project:
  Edit / **Copy** (ctrl+C)
  Edit / **Paste** (ctrl+V)
- to generate simple text reports **File** / **Report / Textual view**
- to print a schema on the printer **File** / **Print**
- to choose and configure the printer **File** / **Printer setup**

• We have produced a new type of file:
  - dictionary reports (`*.dic`).

Exercises for Lesson 2

Finding interesting exercises for such a lesson is quite a challenge! If you insist, try these; otherwise start the next lesson.

Open the `LIBRARY` project (or its French equivalent `BIBLIO`) and its conceptual schema `Library/Conceptual`.

2.1 Examine the semantic description of the objects in the schema. Change and complete some of them.

2.2 Change the position of some attributes and roles in text views. Examine the graphical view and change the position of some objects.

2.3 Find the other side of a rel-type from an entity type.

2.4 Open project `Library` (or its French equivalent `BIBLIO`) and schema `Library/Conceptual`. Generate and print a report based on each of the text views. Try to find specific uses for each of them.

2.5 Open a Text standard report with a text processor. Include after each entity type title the graphical representation of the entity type (through the **Copy graphic** command).

2.6 **Aligning objects**. I'm not quite sure that you have completed the exercise suggested in Figure 2.17! Now it's time to do it.

Lesson 3

Multi-product projects

Objective

This lesson introduces the concept of multi-product projects by considering the example of a design in which we distinguish the conceptual schema and the logical schema of a database as well as two text files. Some characteristics of relational logical schemas are examined. Additional functions related to schema and object management are described as well.

3.1 Starting Lesson 3

We start DB-MAIN, we open the project `MANU-2`, then the schema `Manufacturing/Conceptual`.

3.2 Conceptual and logical schemas

The way we worked in Lesson 1 to produce an SQL database structure was a bit simplistic: we designed a conceptual schema, then we generated the equivalent SQL code to be executed by an RDBMS[1]. This procedure is fine for small databases, but is not realistic for large projects. Of course, it is much too early to tackle the problems induced by managing complex projects, but we

can already introduce the concept of *multi-schema projects*, i.e., projects that include more than one schema, through a more sophisticated procedure than that suggested in Lesson 1.

Let us suppose that we want to keep in the project not only the description of the *conceptual schema* (i.e., the current schema `Manufacturing/Conceptual`), but also the description of the *logical schema*. In traditional database design methododologies, the logical schema is intended to describe the same real-world situation as the conceptual schema does, but in technical terms of tables, columns, primary keys, foreign keys and indexes instead[2]. The logical schema is made up of the database structures that are encoded into a SQL program.

To develop these concepts, we need to go back to the project `Manu-2` that is currently opened.

To give us the opportunity to go through this lesson again later on, we work on a new project called, say, `Manu-3`, which has the same contents as `Manu-2`, at least initially.

To do so, we call the Project property box through the command **File** / **Project properties**, we modify the name into Manu-3, and save the current project (**File** / **Save project as**) as `Manu-3.lun`. From now on, we have two projects, namely `Manu-2`, which is closed and `Manu-3`, the current project on which we will work. So far, these projects have the same contents.

Building a *relational logical schema* is fairly easy, though we may have no idea on how to translate a conceptual schema into relational structures, i.e., into tables, columns, keys and the like. Indeed, DB-MAIN proposes a function which carries out this translation automatically by replacing a schema by its SQL logical equivalent version. Since we want to keep both schemas in the project, we proceed as follows:

1. *Cleaning and modifying the project* `Manu-3`.

   We can get rid of the schema `Alignment`, that is no longer useful. In the same way, we delete the SQL program generated in Lesson 1.

   Deleting objects is quite simple and intuitive: we select the objects, then we press the Del key. Another way is through the command **Edit** / **Delete**.

---

1. Relational Database Management System. DB2, Sybase, Informix, Oracle, SQL Server, InterBase and Access are some examples of RDBMS.
2. See the lessons of [DBM, 1999], or reference textbooks such as [Teorey,1999], [Batini,1992] or [Blaha,1998], [Elmasri, 2000].

Now, the project looks like Figure 3.1.

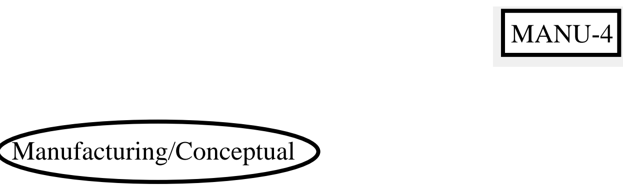


Figure 3.1 - The Manu-3 project in its starting state.

Our conceptual schema is a bit simplistic, and we could find it interesting to enhance it a little. We open the schema, and we state that a *product can be manufactured by an arbitrary number of companies*. Accordingly, we change the cardinality of the role `manufactures.PRODUCT`[3] from `[1-1]` to `[0-N]`. To do so, we double-click on the role and we change the cardinality value, either by typing it or by selecting it in the listbox. The new version should appear as in Figure 3.2.

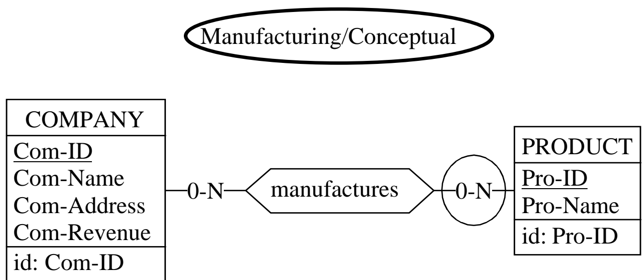


Figure 3.2 - The new `Manufacturing` conceptual schema.

2. *Making a copy of the first schema.*

   Let us make a copy of the conceptual schema:

   - we select the source schema in the Project window, or we open it (it is the current case);

3. A role can be designated by the name of the rel-type followed by the name of the entity type. Another way to denote roles will be seen later.

- we execute the command **Product** / **Copy product**;
- the *Schema property box* opens and proposes default characteristics for the new schema: the name is that of the source schema, "`Manufacturing`", while the version proposed is "`Conceptual-1`". We change the version into "`Relational`" and we click on the button `OK`.

The project window shows the new schema as well as its relationship with the source conceptual schema (Figure 3.3).

We open the so-called *Relational* schema. Not surprisingly, it includes the same objects as the conceptual schema, which is fairly common with copies!

3. *Translating this copy into relational structures.*

Now we transform this schema into relational structures. We execute the command **Transform** / **Relational model**. The contents of the window are replaced by SQL structures. To improve the readability, we shade the "entity types" (through **Views** / **Graphical settings**), now to be interpreted as tables.

If things have gone right so far, the schema `Manufacturing/Relational` should now read as in Figure 3.4.

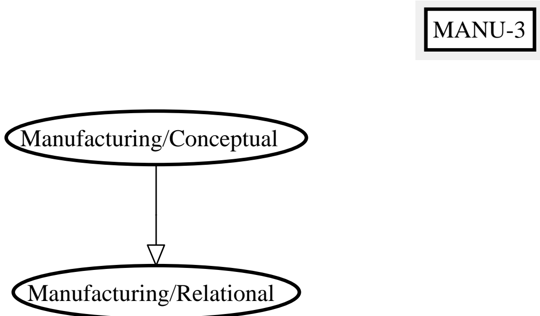


Figure 3.3 - The new *Relational schema* deriving from the *Conceptual schema*.

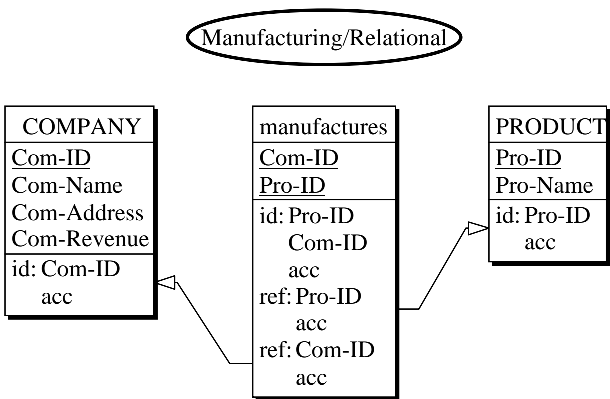


Figure 3.4 - The Relational schema.

This schema is no longer a conceptual schema since it represents data structures of a specific DBMS: instead of *entity types*, we will talk about `tables`, while *attributes* will be called `columns` and *identifiers*, `primary keys`. This kind of schema is called a *relational logical schema*.

The main modification of the schema is the translation of relationship type `manufactures` into entity type `manufactures`.

We observe that the table `manufactures` is made up of the column `Com-ID` which acts as a reference, i.e., a foreign key (ref), to the table `COMPANY`, and of the column `Pro-ID` which references the table `PRODUCT`. Both reference columns form the identifier (i.e., the *primary key*) of the table. In addition, an index (access key or acc) is defined on each identifier and on each reference column to give these structures reasonable performance.

Later on, we will examine in greater detail the way identifiers, foreign keys and indexes are built and represented.

3.3 SQL code generation

Currently, we have two schemas in our project, but still no SQL program that could be used to build the corresponding database in the target computer. Therefore we need a final operation to generate this SQL code. We could use the

command **Transform** / **Quick SQL** as in Lesson 1, but we will explore a more professional way.

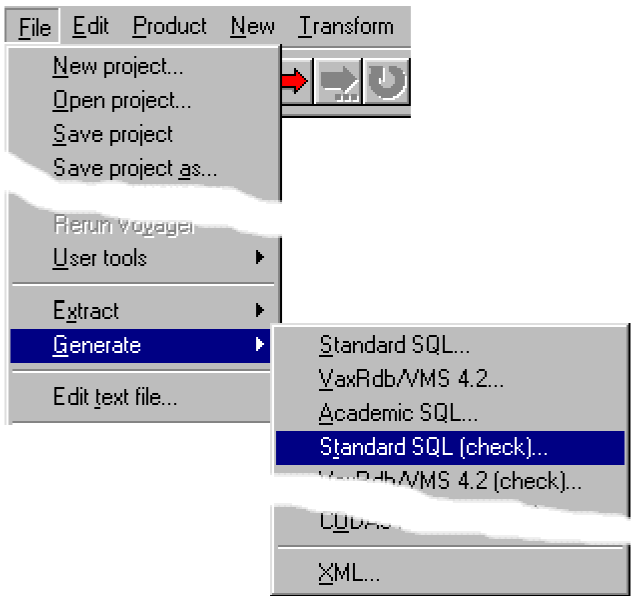


Figure 3.5 - Generating a SQL program from the Relational schema.

We execute the command **File** / **Generate**, then we select the **Standard SQL (check)** style (Figure 3.5). There are other more sophisticated ways to produce SQL code, but for the purpose of this lesson, this style is quite sufficient.

The project windows now shows a new product, namely the file manu-3.ddl which contains the SQL script (Figure 3.6).

By double clicking on the icon of this file, we can examine its contents (Figure 3.7). It is interesting to compare this script with that of Figure 1.19, and to try to understand how the cardinality of the roles of `manufactures` have shaped the resulting logical schema.

This SQL code may not work as such on some DBMS. Indeed, some processing should have been done before generating this text. We will discuss these problems in further lessons.

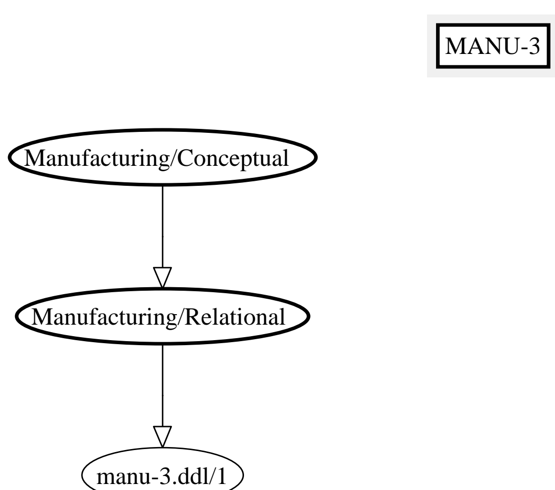


Figure 3.6 - Generation of the SQL script from the relational schema.

```
create database Manufacturing;

create table COMPANY (
     Com-ID char(15) not null,
     Com-Name char(25) not null,
     Com-Address char(50) not null,
     Com-Revenue numeric(12) not null,
     primary key (Com-ID));

create table manufactures (
     Com-ID char(15) not null,
     Pro-ID char(8) not null,
     primary key (Pro-ID,Com-ID));

create table PRODUCT (
     Pro-ID char(8) not null,
     Pro-Name char(25) not null,
     primary key (Pro-ID));

alter table manufactures add constraint FKman_PRO
     foreign key (Pro-ID)
     references PRODUCT;
```

```
alter table manufactures add constraint FKman_COM
     foreign key (Com-ID)
     references COMPANY;

create unique index IDCOMPANY
     on COMPANY (Com-ID);

create unique index IDmanufactures
     on manufactures (Pro-ID,Com-ID);

create index FKman_PRO
     on manufactures (Pro-ID);

create index FKman_COM
     on manufactures (Com-ID);

create unique index IDPRODUCT
     on PRODUCT (Pro-ID);
```

Figure 3.7 - The SQL program. The comment lines and the line numbers have been removed to shorten the listing.

3.4 Generating reports

To complete the project, we generate a report from the conceptual schema just like we done in Lesson 2 (Figure 2.20). Remember that the schema must be shown in text view (button .). When executing the command **File** / **Report** / **Textual view**, we check the button Show report generation to include the icon of the report in the *Project window*. Since any derived product is placed under its source, we could have to move it to a better position (Figure 3.8).

3.5 Multi-product project

So far, our project comprises four documents or *products*, namely two schemas and two text files. A large project can include hundreds of products.

It is sometimes useful to examine two products in parallel. The best way to proceed is as follows:

- open both products,
- minimize the *Project window* (click on the leftmost of the three buttons at the top right corner on the window),

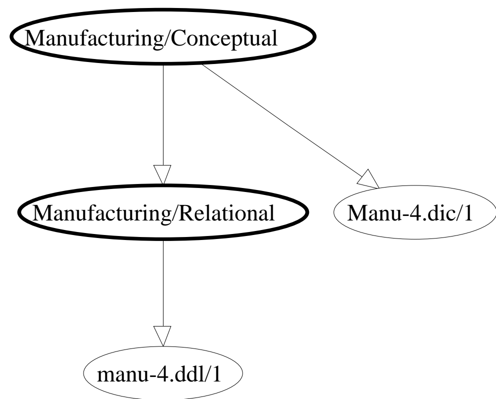


Figure 3.8 - A report has been generated from the conceptual schema.

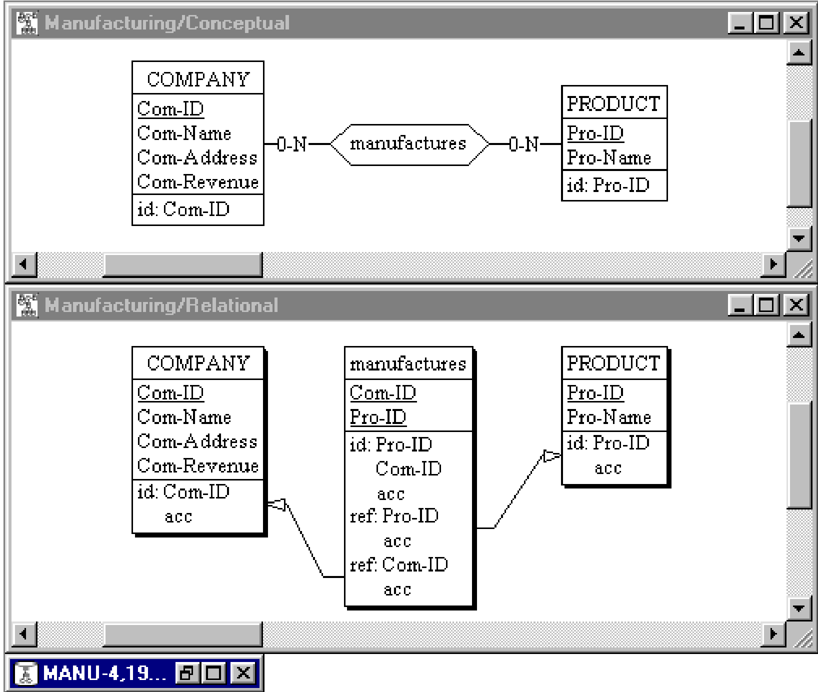


Figure 3.9 - Comparing the conceptual and logical schemas.

- organize the windows by **Window** / **Tile**.

Figure 3.9 shows the conceptual and logical schemas while Figure 3.10 presents the logical schema and its SQL equivalent side by side.

If we want to make the schema disappear from the screen, we can close its *Schema window* by clicking on the close button of that window (the X button at the top right corner). Opening it again can be done by double-clicking on its icon in the *Project window* (Figure 3.8).

3.6 Deleting objects

Deleting components of a project is the simplest thing on earth: we select the objects, then we press the Del key on the keyboard. This applies to entity types, relationship types, roles, attributes, groups (e.g., identifiers), constraints and even schemas. An alternate way consists in executing the command **Edit** / **Delete**.

There is no way to delete a project but by deleting its `*.lun` file from Windows.

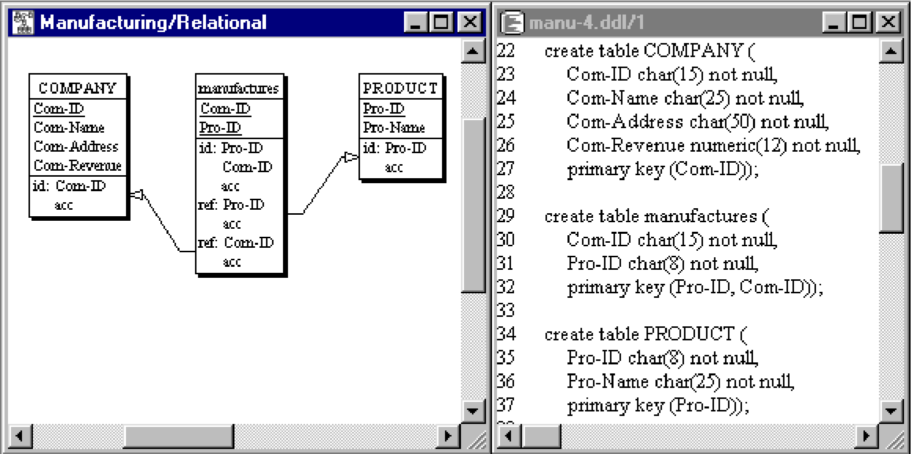


Figure 3.10 - Comparing the logical schema with its SQL text.

3.7 Quitting the lesson

We can now quit DB-MAIN through command **File** / **Exit**. The modified project can be saved as suggested by DB-MAIN.

Summary of Lesson 3

- In this lesson, we have studied the following concepts:
- conceptual and logical schemas
- products, which are either schemas or text files,
- multi-product projects

- We have also learnt:
- to create and use a multi-product project
- to make a copy of a product: **Product** / **Copy product**
- to transform a schema **Transform** / **Relational model**
- to generate SQL code **File** / **Generate**
- to delete an object **Edit** / **Delete** or Del key
- to arrange the schema windows: **Window** / **Tile**

Exercises for Lesson 3

3.1 Open the project `SALES1` you built as a solution to Exercise 1.2. Complete this project by building a relational logical schema, and by generating a SQL program. Examine the schemas side by side, and compare them.

Can you understand some of the rules that have been applied during the schema transformation? If you don't, never mind, we will study them in detail later on.

3.2 Same exercise on project `STUDENT1` of Exercise 1.3.

3.3 Same exercise on project `LIBRARY` (or its French version `BIBLIO`). Make sure you don't save the result inadvertently, except through a **Save as** command.

Lesson 4

Conceptual Modeling

Objective

This lesson will introduce the reader to more powerful features of the DB-MAIN conceptual model. In particular, he will learn to define optional/mandatory attributes, atomic/compound attributes, single-valued/multivalued attributes, multiple identifiers, hybrid identifiers, N-ary relationship types, relationship types with attributes, and others with identifiers, cyclic relationship types.

Preliminary checking. In this lesson, we will use project `MANU-3` (file `manu-3.lun`) that has been created in Lesson 3.

4.1 Starting Lesson 4

We start DB-MAIN and we open the project `MANU-3`. We delete all the products but the Conceptual schema. We rename the project `MANU-4` and we save it under the name `MANU-4.lun` (**File** / **Save project as**). We open the schema `MANUFACTURING/Conceptual`.

4.2 Updating an object

We have seen in Lesson 3 how to update the properties of a schema (namely its Version). This technique also applies to any object of a project:

- either double-click on the object name in its Schema window, or select the object (by clicking on its name) and press the `RETURN` key; either of these actions opens the object box;
- change the concerned properties of the object;
- either validate the operation by clicking on the `OK` button, or discard the modifications by clicking on the `Cancel` button.

This works fine for schemas, entity types, relationship types, attributes and groups[1]. The only exception is the project itself, that only appears as a passive object on the screen[2]. To modify its properties, use the command **File**/ **Project properties** instead.

4.3 What is a conceptual schema?

Despite its limited scope, Lesson 1 has introduced some important notions about conceptual schemas. First, it showed that such schemas are technology-independent in that they comprise abstract objects that denote application domain concepts independently of their representation through DBMS cons-

1. ... and for collections, constraints, etc, as we will see later on.
2. In fact it is less passive than it seems to be, but we shall ignore its behaviour in these lessons.

tructs. The schema of Figure 4.1 has been developed by the analysis of the facts the application domain is made up of. The way these facts will be represented in terms of tables, columns and foreign keys is irrelevant at this stage.

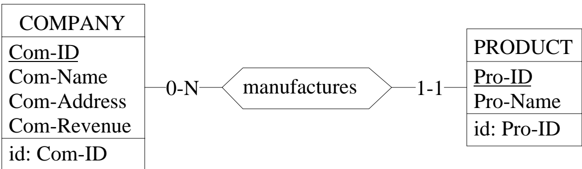


Figure 4.1 - The conceptual schema we built in Lesson 1.

This first experiment has taught us that a conceptual schema comprises entity types (`COMPANY`, `PRODUCT`), relationship types (`manufactures`), attributes (`Com-ID`, `Pro-Name`) and identifiers ({`Com-ID`}, {`Pro-ID`}).

An **entity type** represents a class of similar objects, or entities, that are perceived as significant when we talk about the application domain. Such objects are modeled through an entity type when we want to record information about them, when they are associated with other entities and when they obey to specific behaviour rules.

A **relationship type** (rel-type) models similar associations between the entities of two entity types. A relationship is a pair of entities, each of them belonging to one of the participating entity types. Each participating entity type plays a definite **role** in the rel-type. This role is characterized by a cardinality constraint expressed as a pair of symbols such as [1-1] or [0-N].

An **attribute** denotes a property of an entity type. It has a type (numeric, character, date, etc.), a length and a cardinality.

An **identifier** is a group of attributes that uniquely qualifies the entities of a type. At any time, two entities of this type must have distinct values for the attributes of the identifier.

In this lesson, we will discuss variants of these concepts as well as new concepts that will be useful to build more expressive conceptual schemas.

4.4 Cardinality of an attribute

Until now, we have implicitly considered that each entity of a given type had one and only one value for each of its attributes: each `COMPANY` entity has one value of `Com-ID`, one value of `Com-Name`, one value of `Com-Address` and one value of `Com-Revenue`.

We now consider that this is not true for the latter attribute: some companies have revenues while others may have none. Therefore, some `COMPANY` entities have one value of `Com-Revenue`, while others have none. In general, we can say that any `COMPANY` entity has from 0 to 1 `Com-Revenue` value and from 1 to 1, i.e., exactly one, `Com-Name` value.

The values 0-1 and 1-1 are called the **cardinality** of the attribute. Any couple of non-negative values is valid, provided the first one is not greater than the second one and the second one is at least 1. The default value is 1-1, and is not displayed in the Schema windows.

To illustrate this concept, we double-click on the attribute `Com-Revenue` to call its Attribute box, and change its cardinality from 1-1 to 0-1. Then, we define a new attribute, named `Phone-Number`, that is given cardinality 1-4, stating that any company has from 1 to 4 phone numbers (Figure 4.2).

```
COMPANY
-------------------
Com-ID
Com-Name
Com-Address
  Number
  Street
  City
    Zip-Code
    City-Name
Com-Revenue[0-1]
Phone-Number[0-4]
  Country
  Area
  Local
-------------------
id: Com-ID
```

Figure 4.2 - Various kinds of attributes.

4.5 Mandatory and optional attributes

An attribute whose cardinality has a lower bound of 0 is called **optional**. Conversely, an attribute whose cardinality has a non-zero lower bound is called **mandatory**. For instance,

- `Com-Revenue` is optional,
- `Com-Name` is mandatory,
- `Phone-Number` is mandatory.

4.6 Single- and multivalued attributes

An attribute whose cardinality has an upper bound greater than 1 is called **multivalued**, while those with cardinality 0-1 or 1-1, are said to be **single-valued**. For instance,

- `Phone-Number` is multivalued,
- `Com-Name` is single-valued.

4.7 Atomic and compound attributes

Some attributes can be broken down into fragments that still are significant. For instance, any value of `Com-Address` can be seen as composed of a value of `Number` + a value of `Street` + a value of `City`.

`Com-Address` is a **compound** attribute and `Number`, `Street` and `City` are its components. Note that a component can itself be compound; such is the case of `City`, which consists of `Zip-Code` and `City-Name`.

An attribute that is not compound is called **atomic** (i.e., *unbreakable*). For instance, `Com-Name`, `Com-Address.Number`[3] and `Com-Address.City.City-Name` are atomic attributes.

Both single-valued (`Com-Address`) and multivalued (`Phone-Number`) attributes can be compound.

Changing `Com-Address` from atomic to compound cannot be easier:

- we select attribute `Com-Address` then we click on button in the

3. This notation designates the component `Number` of the compound attribute `Address`.

Standard tool bar;

- we define its first component, Number;
- we click on the button Next att. in the Attribute property box to define the other components.

We modify the structure of COMPANY as shown in Figure 4.2.

Note that a compound attribute has a length too. However, this length is calculated, and cannot be changed through the Attribute box itself.

You probably have observed that entity types and relationship types also are assigned a length field. Its value is the sum of the lengths of their attributes, if any.

4.8 Multiple identifiers

An entity type can have more than one identifier. For instance, the entity type COMPANY is identified by Com-ID, which means that, in the database described by the schema, no two COMPANY entities will be allowed to share the same value of Com-ID.

In addition, let us assume that there are no two companies with, simultaneously, the same name and the same address. Therefore, we will specify a second identifier, comprising Com-Name and Com-Address: we select both attributes, then we proceed as in Lesson 1, by clicking on button ID in the Standard tool bar. The new identified appears with prefix id'.

If an entity type has identifier(s), one of them generally is declared *primary* (notation id), while the others, if any, are declared *secondary*, and are noted id' instead. Such is the case of the new identifier (Figure 4.3).

Note that an entity type can have secondary identifiers only. However, it can have only one primary id. It is a good practice to define the most natural identifier as primary. The problem of choosing identifiers can be a bit more complex, and will be discussed later on.

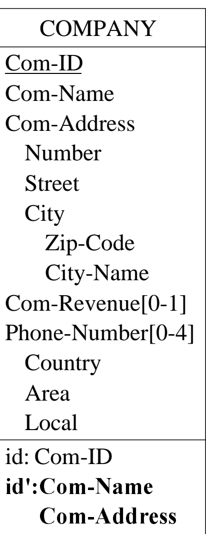


Figure 4.3 - Primary (id) and secondary identifiers (id') of an entity type.

4.9 Hybrid identifiers

Until now, the identifiers have been made up of one or several attributes of their entity type. In some situations however, an identifier can be more complex.

To illustrate this point, we need a more sophisticated schema. We suppose that a company comprises at least one branch, and that branches, not companies, manufacture products. Therefore:

- we create the entity type `BRANCH`, with attributes `Name` and `Country` (button);
- we create the one-to-many relationship type `belongs` between `BRANCH` and `COMPANY` (button);
- in `manufactures`, we replace `COMPANY` with `BRANCH` (we delete the old role then we draw a new arc with button).

In addition, let us suppose that all the branches of a company are located in distinct countries.

Such a situation can be described by stating that a `BRANCH` entity is identified by its `COMPANY` (via `belongs`) + its `Country`. An identifier made up of attributes and roles, is called *hybrid*. By extention, we will call **hybrid** any identifier comprising at least one role.

A hybrid identifier is defined in the same way as *all-attribute* identifiers: by selecting the components, be they attributes or roles then by clicking on button ID.

We complete the schema as shown in Figure 4.4.

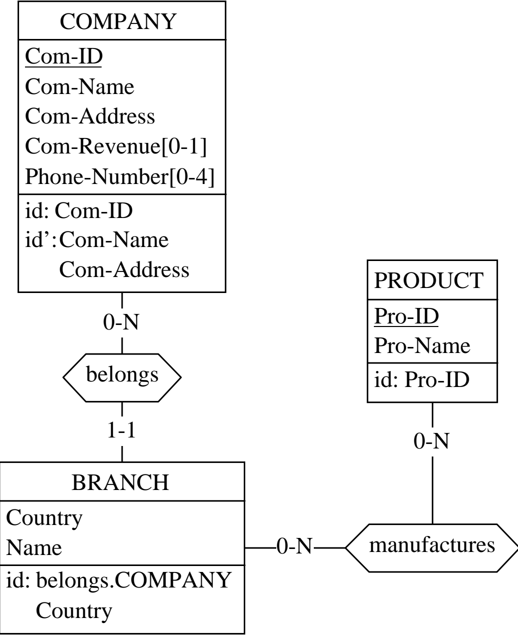


Figure 4.4 - Hybrid identifier. To save space, the components of the compond attributes have been ignored.

In short, the identifier of an entity type can be made up of:

- either *any number of attributes* (at least 1);
- or *one role* + *any number* (at least 1) *of attributes and/or roles*.

Any non empty combination of roles and attributes is allowed, except that which is made of one role only. The concept of identifier can be more complex than it appears in this lesson. It will be discussed in detail in a future lesson.

Last remark. An entity type need not have identifiers. Entity types without identifiers are unfrequent, but quite valid however.

4.10 N-ary relationship types

The relationship types we have defined so far are made up of two roles. It is possible to define relationship types with three (or more) roles.

In the following schema, we have defined a new entity type, namely `MARKET`, that represents the different markets on which products can be distributed. We give it the attributes `Name` and `Size`. In addition, we have considered that a branch manufactures products for some markets only. Therefore, a `manufactures` relationship links one `BRANCH` entity, one `PRODUCT` entity and one `MARKET` entity. Such relationship expresses the fact that *this branch* manufactures *this product* for *this market*[4].

We can change the relationship type `manufactures` from binary (2 roles) to ternary (3 roles) as follows:

- we draw an arc (button) from `manufactures` to `MARKET`;
- if needed, we change the cardinality and the name of the new role.

In general, non-binary relationship types are called **N-ary**, where N is the number of roles. The resulting schema is proposed in Figure 4.5.

4.11 Relationship types with attributes

Attributes can be associated to relationship types as well. Let us suppose that the manufacturing of a product by a branch for a given market is measured by a ratio.

The attribute describing this ratio is created in the same way as for entity types:

- we select `manufactures` by clicking on its name;
- we click on the button in the Standard tool bar and we define the attribute.

4. We suppose that there is no constraint on the possible associations between branches, products and markets. For instance, a branch can manufacture a product for some markets, and another one for other markets. In technical terms, specialists will say that there is no dependencies holding in this relationship type. More on this later on.

The relationship type `manufactures` should look like in Figure 4.6.

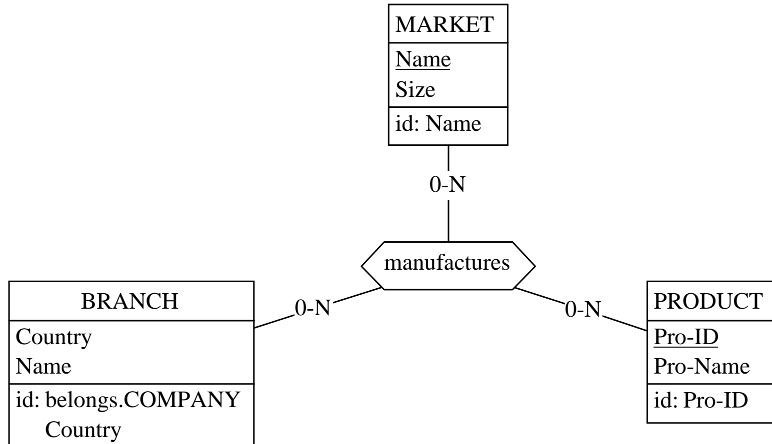


Figure 4.5 - An N-ary relationship type. To save space, the attributes have been ignored.

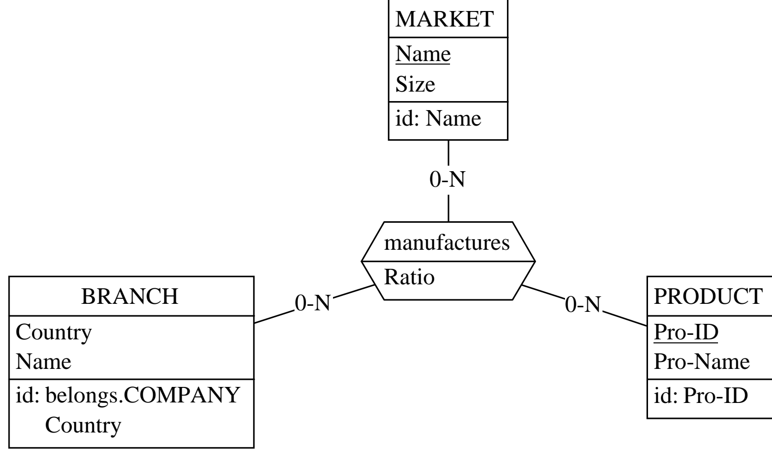


Figure 4.6 - A relationship type with an attribute.

4.12 Relationship types with identifier(s)

Relationship types can have identifiers too. For instance, we could consider a new rule stating that

when a branch manufactures a product, it does it for one market only.

This property can be expressed by an identifier of `manufactures` comprising `PRODUCT` and `BRANCH`. If we designate a `PRODUCT` entity and a `BRANCH` entity, the database can include only one `manufactures` relationship in which they both appear, and therefore only one `MARKET` entity.

Such an identifier is defined in a more complex way than for entity types[5]:

- we select `manufactures` by clicking on its name;
- we execute the command **New** / **Group**; we define a group as *Primary* (by clicking on the buton `Id`), and we move components `PRODUCT` and `BRANCH` from the right list to the left list (Figure 4.7).

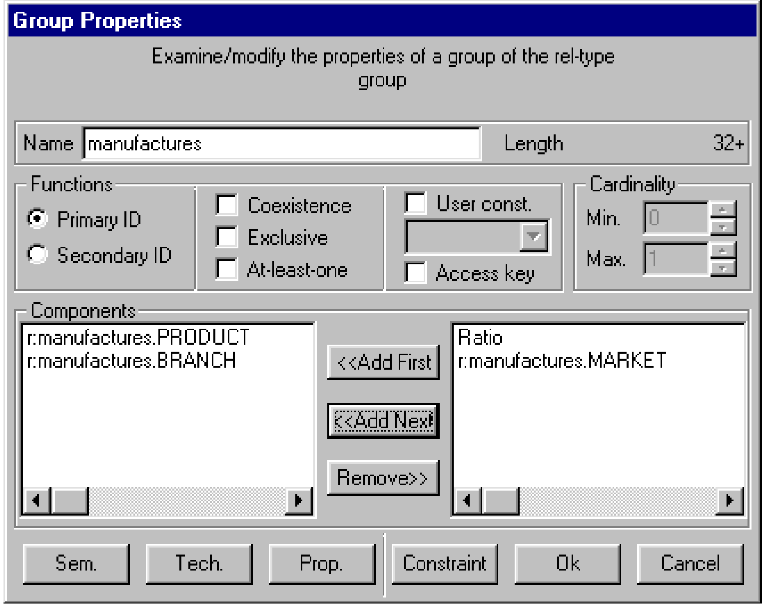


Figure 4.7 - Defining an identifier as a group of roles.

5. In fact, this technique is the standard way to define any identifier, but so far, we have used a simpler one which is valid for entity types only.

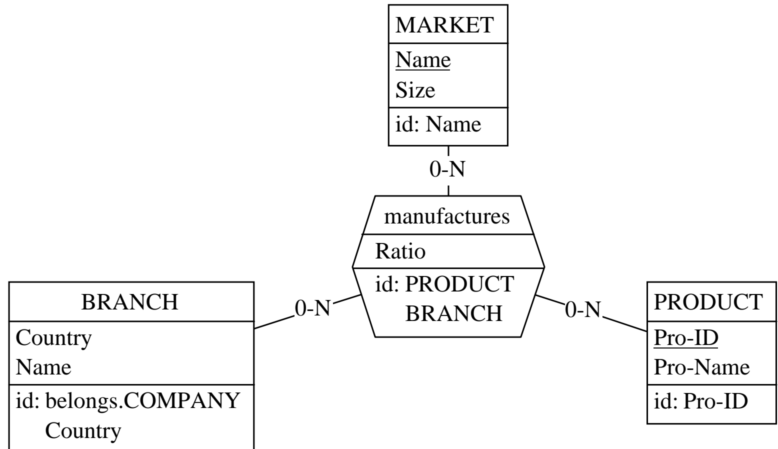


Figure 4.8 - Relationship type identifier.

The new version of `manufactures` is shown in Figure 4.8.

In fact, every relationship type has (at least) one identifier, but most of them should not be declared explicitly as illustrated hereabove. DB-MAIN will consider as an implicit identifier of relationship type R,

- each role of R with cardinality 0-1 or 1-1,
- all the roles of R when there are no such 0-1 or 1-1 role, and when no explicit identifier is declared.

For instance, the (implicit) identifier of relationship type `BELONGS` is `BRANCH`, and the (implicit) identifier of `MANUFACTURES` in `MANU-1` is `(COMPANY,PRODUCT)`[6]. Therefore, such identifiers need not be declared, DB-MAIN being able to cope with them adequately.

6. According to the maximum cardinality of 1, there is only one `belongs` relationship in which a given BRANCH entity appears. Concerning the `manufactures` relationship type, there is no need to state more than once that a given company manufactures a given product, hence the rule.

4.13 Cyclic relationship types

Each role of a relationship type is defined as the participation of an entity type. A relationship type in which the same entity type participates more than once is perfectly valid.

Let us consider that a product can be replaced, when unavailable, by another product. This fact can be represented easily by relationships between some PRODUCT entities and other PRODUCT entities. Such relationships form a **cyclic** relationship type.

To represent this, we define a new relationship type, with name replaces, and with two roles, both defined on PRODUCT, with cardinality 0-1 and 0-N respectively. To distinguish the function of each of these roles, we will give them distinct names. The role corresponding to the product that is replaced will be called replaced, while the role corresponding to the product that replaces the former will be called substitute (Figure 4.9).

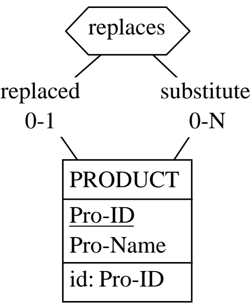


Figure 4.9 - A cyclic relationship type.

About role names

Until now, we have ignored the names of roles. When we give a role no name, DB-MAIN gives it as default name that of the participating entity type. For instance, the belongs relationship type has two roles with default names COMPANY and BRANCH, though we gave them no explicit names.

This being said, we can state a property each relationship type must satisfy: *its roles have distinct names*, be they explicit or default. Applying this property to cyclic relationship types means that all the roles played by the same entity type (more precisely all of them but one) must receive an explicit name which is different from that of the entity type.

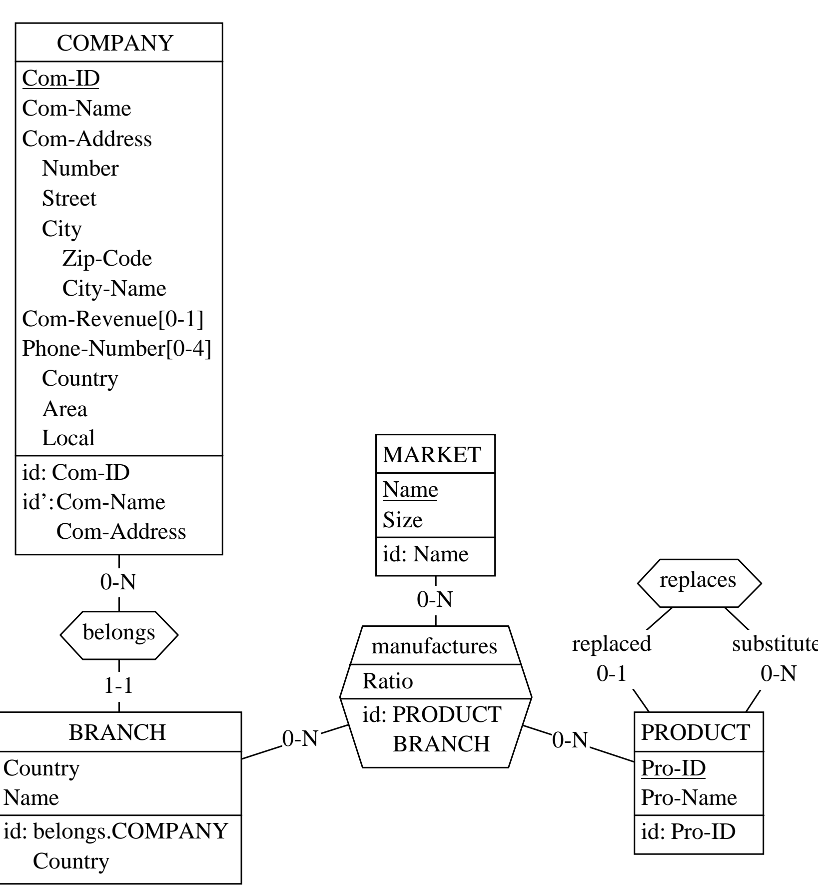


Figure 4.10 - The complete schema.

Since the same role name may appear in several relationship types, its name alone cannot identify it in its schema. Therefore, the full name of a role includes also that of its relationship type. For instance, the roles of `BELONGS` have full names `belongs.BRANCH` and `belong.COMPANY`, and those of `replaces` have names `replaces.substitute` and `replaces.replaced`. Accordingly, these full names appear in the list boxes of the Group boxes and in the specification of the groups in the schemas.

```
                  Schema Manufacturing/Conceptual-Final

         BRANCH
            Country
            Name
            id: belongs.COMPANY,Country
         PRODUCT
            Pro-ID
            Pro-Name
            id: Pro-ID
         COMPANY
            Com-ID
            Com-Name
            Com-Address
               Number
               Street
               City
                  Zip-Code
                  City-Name
            Com-Revenue[0-1]
            Phone-Number[1-4]
               Country
               Area
               Local
            id: Com-ID
            id': Com-Name,Com-Address
         MARKET
            Name
            Size
            id: Name

         manufactures (
             [0-N]: BRANCH
             [0-N]: PRODUCT
             [0-N]: MARKET
            Ratio )
            id: PRODUCT,BRANCH
         belongs (
             [0-N]: COMPANY
             [1-1]: BRANCH )
         replaces (
            substitute [0-N]: PRODUCT
            replaced [0-1]: PRODUCT )
```

Figure 4.11 - The complete schema in text view.

4.14 The complete schema

If all the extensions described above have been included, the schema should appear as in Figure 4.10 or as in Figure 4.11 in the *Text standard* view.

4.15 Quitting the lesson

This lesson is finished. We save the current project and we quit DB-MAIN.

Summary of Lesson 4

- In this lesson, we have studied the following concepts:
- the cardinality of an attribute
- single-valued / multivalued attributes
- mandatory / optional attributes
- atomic / compound attributes
- multiple identifiers
- hybrid identifiers
- N-ary relationship types
- attributes of relationship types
- identifiers of relationship types
- cyclic relationship types
- role names.

- We have also learnt to:
- update the properties of an object
  double-clik on the object description
  or **File** / **Project properties**
- define the cardinality of an attribute
- define a compound attribute
  button
- add a role to a relationship type
  button
- add attributes to a relationship type
  button
- define an id. for a relationship type
  New / **Group**
- give a name to a role

Exercises for Lesson 4

4.1 Build a schema describing persons who have each a person id, a name, 1 to 3 christian names, possibly a maiden name, and an arbitrary number of addresses.

4.2 These persons may have children, who are persons too.

4.3 Build a schema which represents customers, products and suppliers (with some natural properties such as name, address, quantity on hand, etc). Represent the fact that suppliers supply products to customers, and that they do so in a given supplied quantity.

Lesson 5

Logical and Physical Modeling

Objective

The 5th lesson discusses some concepts of the DB-MAIN model dedicated to the representation of technical constructs, i.e., components that appear in DBMS schemas as opposed to those that make up computer-independent conceptual schemas. We will describe and manipulate additional integrity constraints (e.g., referential constraints), access keys (representing indexes for instance) and entity collections (representing record files). We will also examine how to transform the names in a schema.

Preliminary checking. In this lesson, we will use the project MANU-4 (file manu-4.lun) that has been created in lesson 4.

5.1 Starting Lesson 5

We start DB-MAIN and we open the project MANU-4. As usual, we change its name (MANU-5) and we save it (manu-5.lun).

In this project, we then open the schema Manufacturing/Conceptual.

5.2 What is a logical schema?

Lesson 3 shown how a conceptual schema can be translated into a relational schema. Both schema represent the same information, but the latter expresses it through the constructs of a DBMS[1], while the former is claimed to be DBMS-independent. A relational schema is considered to be **logical**. The same conceptual schema can be transformed into several relational logical schemas, according to the design criteria we have in mind: readability, simplicity, ease of evolution, response time, space occupied on disk, etc. To keep things simple, we will mainly concentrate on **relational schemas**, i.e., on logical schemas that comply with the relational model.

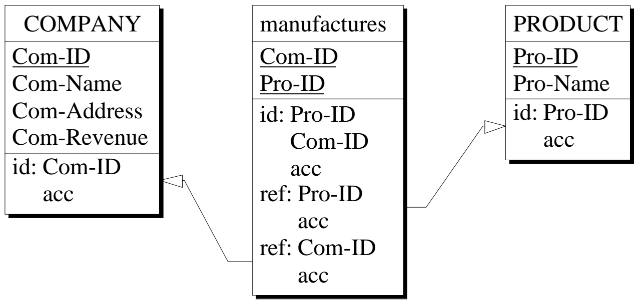


Figure 5.1 - The logical schema built in Lesson 3.

1. In other words, the conceptual structures are expressed into the model of a DBMS, or, more precisely, into the model of a family of DBMSs.

A relational logical schema comprises *tables* made up of *columns*, *primary* (or *secondary*) *keys* and *foreign key*. Figure 5.1 shows the logical schema we built in Lesson 3. It includes three tables, eight columns, three primary keys (id), two foreign keys (ref). In addition, it includes indexes (acc, for access keys), which have been defined on each key.

In this lesson, we will discuss in greater detail the concepts of which all relational logical schemas are made up.

5.3 Transformation into a logical schema

Let us produce a relational logical schema for the conceptual schema we developed in Lesson 4. We proceed as suggested in Lesson 3:

- we make a copy of the schema (we select schema `Manufacturing/Conceptual` then execute **Product** / **Copy schema**) and we change its version value to "`Logical`";
- in this new schema, we execute **Transform** / **Relational model** to produce the relational structures;
- we change the graphical representation by adding shade to the entity types (**View** / **Graphical settings**), to make them look like tables[2] (with a little imagination!).

Schema `Manufacturing/Logical` is transformed into *relational data structures* (Figure 5.3 and Figure 5.4).

From now on, we should use the terms *table* instead of entity type, *column* instead of attribute, etc. However, the logical model is independent of specific technologies, and in particular of relational DBMS. Figure 5.2 gives the translation rules for RDBMS. Similar tables can be built for other data management systems. We will keep using the standard terms of *entity types* and *attributes*, except when mentioned otherwise.

This schema is inevitably more complicated and less readable than its conceptual counterpart (otherwise it would have been preferable to reason from the beginning in the relational model!). The objective of this lesson is not to describe in detail how and why the transformation was carried out. Therefore, we just have to accept this schema as it is.

2. The idea is that shading gives the objects a 3D look, which makes them more *concrete*.

DB-MAIN concepts	Relational terms (SQL)
entity type	table
attribute	column
primary identifier	primary key
secondary identifier	candidate key (*not pure SQL*)
reference group	foreign key
access key	index
entity collection	(table-/db-)space (*not standard*)

Figure 5.2 - Translation table of DB-MAIN names into relational names.

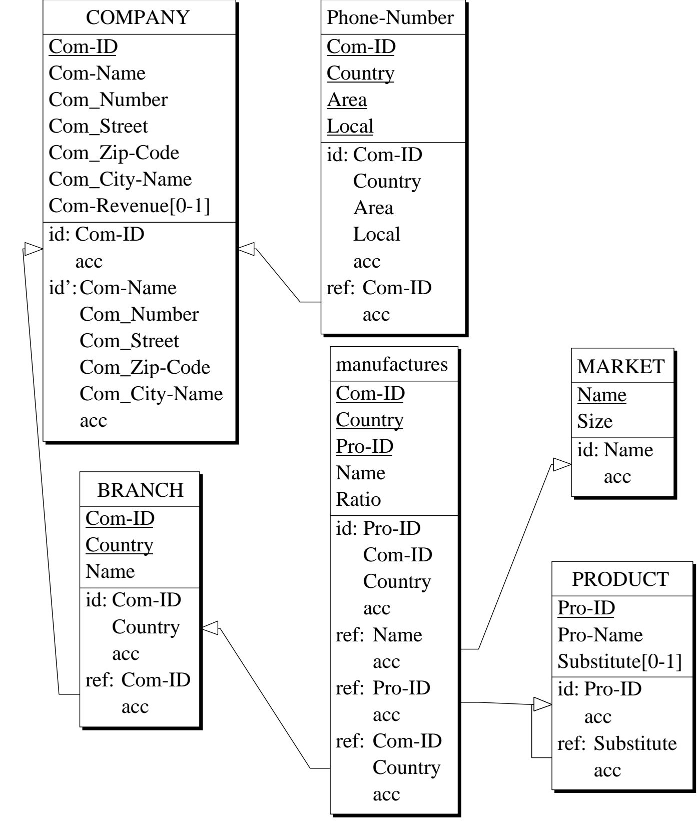


Figure 5.3 - First version of the logical schema.

```
              Schema Manufacturing/SQL

BRANCH
   Com-ID
   Country
   Name
   id: Com-ID,Country
      access key
   ref: Com-ID -> COMPANY.Com-ID
      access key

COMPANY
   Com-ID
   Com-Name
   Com_Number
   Com_Street
   Com_Zip-Code
   Com_City-Name
   Com-Revenue[0-1]
   id: Com-ID
      access key
   id': Com-Name,Com_Number,Com_Street,Com_Zip-Code,Com_City-Name
      access key

manufactures
   Com-ID
   Country
   Pro-ID
   Name
   Ratio
   id: Pro-ID,Com-ID,Country
      access key
   ref: Name -> MARKET.Name
      access key
   ref: Pro-ID -> PRODUCT.Pro-ID
      access key
   ref: Com-ID,Country -> BRANCH.(Com-ID,Country)
      access key

MARKET
   Name
   Size
   id: Name
      access key
```

```
Phone-Number
   Com-ID
   Local
   Area
   Country
   id: Com-ID,Local,Area,Country
      access key
   equ: Com-ID = COMPANY.Com-ID
      access key

PRODUCT
   Pro-ID
   Pro-Name
   Substitute[0-1]
   id: Pro-ID
      access key
   ref: Substitute -> PRODUCT.Pro-ID
      access key
```

Figure 5.4 - First version of the logical schema - Text standard view.

Now, we will discuss in greater detail some important constructs that we already encountered in lesson 3, and that appear again in this schema, namely the *reference attributes* and the *access keys*.

5.4 Reference attributes (foreign keys)

A **reference attribute** is an attribute whose values act as references to other entities. For instance, attribute `Com-ID` in entity type `BRANCH` is aimed at designating a `COMPANY` entity. Since each entity type represents a table in this logical SQL schema, `Com-ID` is what is called a **foreign key** in the RDBMS language. In general, since a foreign key can comprise more than one attribute, we will talk about **reference groups**.

The way this attribute is denoted in DB-MAIN views expresses that each value of `Com-ID` in any `BRANCH` entity must be a `Com-ID` value in some `COMPANY` entity. We observe that the attribute mentioned in the target entity type (here `COMPANY`) is its primary identifier. In some situations, the target attribute can be a secondary identifier as well.

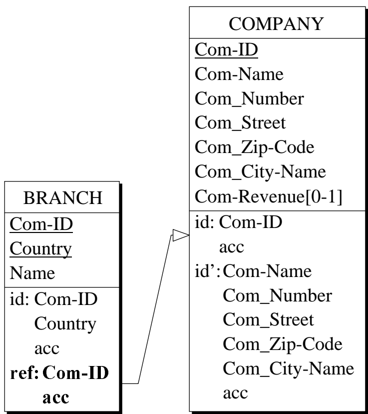


Figure 5.5 - Reference group, *aka* foreign key.

If the identifier of the target entity type is made of several attributes, then the reference must be supported by several reference attributes, as in `manufactures` entity type, where the values of attributes (`Com-ID`, `Country`) designate a `BRANCH` entity (Figure 5.6).

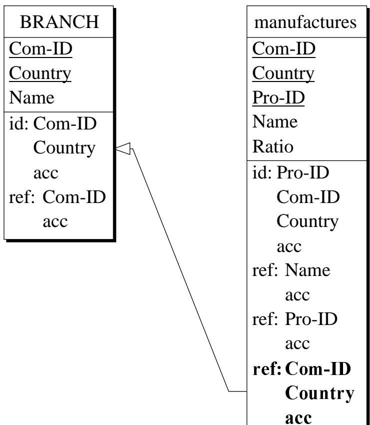


Figure 5.6 - Multicomponent reference group.

5.5 *Equality* reference

There is a more sophisticated form of reference attributes that can be found in entity type (i.e. table) Phone-Number. Let us first observe that each entity of this type represents a phone number of a company, and that all the phone numbers of company X are represented by the Phone-Number entities with Com-ID = X. Therefore, Com-ID is a reference attribute (or foreign key) to COMPANY.

However, the conceptual schema tells us that each company must have **at least one** phone number (cardinality [1-4]). This property translates, in the current logical schema, into a constraint stating that each COMPANY entity must have at least one corresponding Phone-Number entity. More precisely, the value of Com-ID of each COMPANY entity must match the Com-ID value of at least one Phone-Number entity.

Since the COMPANY.Com-ID values form a subset of the PHONE-NUMBER.Com-ID values and the PHONE-NUMBER.Com-ID values form a subset of the COMPANY.Com-ID values, we can conclude that,

the set of COMPANY.Com-ID values ***is equal to*** *the set of PHONE-NUMBER.Com-ID values.*

To represent this constraint, DB-MAIN uses the term equ, that expresses that the value sets of Com-ID in both entity types are ***equal*** (Figure 5.7).

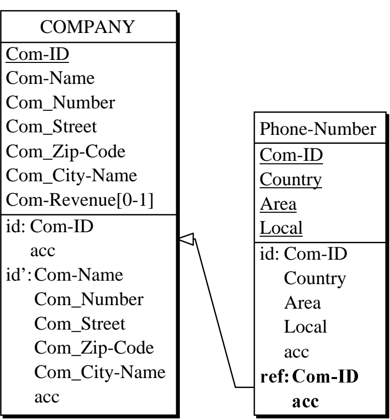


Figure 5.7 - Equality reference group.

5.6 Defining a foreign key

So far, referential attributes are automatically defined as the representation of relationship types. Later on we could find it useful to define referential constraints manually, for instance to document an existing SQL database.

To practice defining referential attributes, we delete the last constraint of entity type `manufactures` by clicking on the "`ref: Com-ID,Country`" line, and pressing the Del key. The line disappears.

To build it again, we define for entity type (table) `manufactures`, a group of attributes comprising `COM-ID` and `Country`:

- we select both attributes, and we click on the button in the *Standard tools bar* (Figure 5.8);
- we open the Property box of this group (press the Enter key or double-click) (Figure 5.9).

Now we have to tell DB-MAIN that this group is a reference to table `BRANCH`. We click on the Constraint button (for *inter-group constraint*). The Constraint box opens (Figure 5.10). We have two properties to specify:

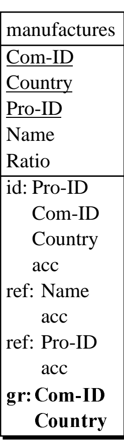


Figure 5.8 - A group comprising {`Com-ID,Country`} has been defined

- what *kind of constraint* do we want? Let us click on the Ref button;
- what is the *target entity type*, and what is the *target identifier*? DB-MAIN will help us considerably by suggesting candidate entity types, and for each of them suggesting candidate identifiers. These suggestions are based on

the structure of the source group we have built, i.e., its composition, the type and the length of its components. In this case, there is not much choice: only the BRANCH entity type has an identifier composed of two attributes whose types and lengths match those of the current group. Therefore, DB-MAIN proposes this target entity type and this identifier only.

To make this schema equivalent to its former version, don't forget to click on the Access key button as well (more on this below).

5.7 Access keys

The transformation has generated *access keys*. This term designates technical data structures that provide efficient selective access to data records. An access key will generally be implemented as an *index* or a *hash table* in relational DBMS. However, the term *access key* has been chosen instead of *index* since each DBMS generally proposes its own names to denote these techniques[3].

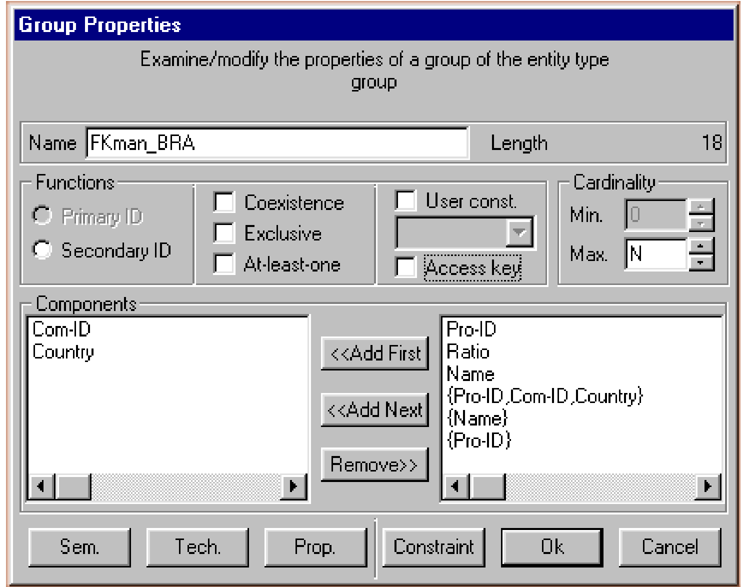


Figure 5.9 - The properties of the newly defined group.

3. Let us cite *record keys* in COBOL files and *calculated keys* in CODASYL databases.

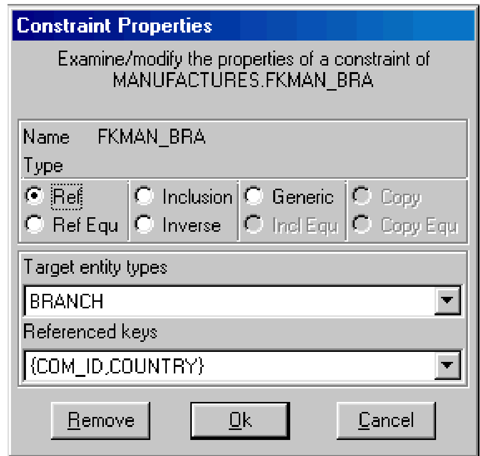


Figure 5.10 - Choosing the target of the reference group.

Let us consider entity type `MARKET` (Figure 5.11). Its attribute `Name` is declared both **identifier** (`id`) and **access key** (`acc` or `access key`). Indeed, RDBMS generally require that each identifier be an index as well. This means that `Name` is an identifier, and, in addition, an access key. Therefore, asking for the `MARKET` whose `Name` is known will lead to a quick answer from the database.

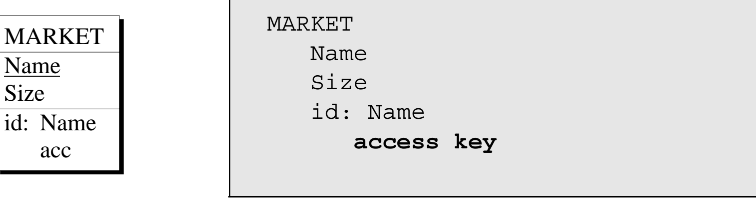


Figure 5.11 - Identifiers often are supported by access keys.

In addition, all the reference groups (foreign keys) have been made access keys as well (Figure 5.12). It is not mandatory, but DB-MAIN has found it handy to propose this in its transformation process. Indeed, such attributes implement relationship types, and therefore will most probably be used as selection criteria in the programs (in join-based queries for instance).

So far, an access key is just an additional property of another construct (identifier or referential group). We can also decide to declare other access keys if we think they can boost the performance of queries.

For instance, we can consider that asking for a product of which only the name is known, is a frequent query. To accelerate the processing of this query, we decide to build an access key on `Pro-Name` of `PRODUCT`.

```
BRANCH
Com-ID
Country
Name
id: Com-ID
    Country
    acc
ref:Com-ID
    acc
```

```
BRANCH
   Com-ID
   Country
   Name
   id: Com-ID, Country
      access key
   ref: Com-ID -> COMPANY.Com-ID
      access key
```

Figure 5.12 - Reference groups (foreign keys) are supported by access keys.

An access key is just a special kind of a *group*. To illustrate it, we add a new group to `PRODUCT`:

- we select attribute the `Pro-Name`, and we click on the button GR ;
- we open the Property box of this group (press the Enter key or double-click) (Figure 5.9);
- we click on the button `Access key` and we confirm the operation.

The entity type `PRODUCT` now reads as in Figure 5.13.

```
PRODUCT
Pro-ID
Pro-Name
Substitute[0-1]
id: Pro-ID
    acc
ref: Substitute
    acc
acc: Pro-Name
```

```
PRODUCT
   Pro-ID
   Pro-Name
   Substitute [0-1]
   id: Pro-ID
      access key
   ref: Substitute -> PRODUCT.Pro-ID
      access key
    access key: Pro-Name
```

Figure 5.13 - An additional access key.

Some access keys enjoys an interesting property, provided they are based on a special kind of implementation, called B-trees[4]:

> *An access key defined on any prefix of another access key can be discarded.*

Let us suppose that an access key has been defined on <A, B, C>. According to this property, candidate access keys <A,B> and <A> are useless, and can be removed from the schema. Indeed, the DBMS can use the first access key to answer queries that would use any of the prefix access keys, without any penalty. This removing can be considered as a simple but efficient optimization technique.

As an application of this technique, we will remove the three prefix access keys of the logical schema (compare the final schema of Figure 5.19 with that of Figure 5.3).

5.8 Defining entity collections

In a real database, that is, one which is implemented in an actual computer, table rows and records are stored in a large secondary memory, such as on a magnetic disk. More specifically, they are stored in storage units called, depending on the data management system, *files, data files, datasets, areas, realms, DBspaces* or *tablespaces*.

DB-MAIN proposes a concept to represent such storage units, namely the **entity collection**, or, more simply, the **collection**.

Let us suppose that the six tables of the relational database have to be stored into two distinct files, one, called `PR_STORE`, which can accomodate the rows of `PRODUCT`, `MARKET` and `manufactures`, and the other, called `CY_STORE`, in which the rows of `COMPANY`, `BRANCH` and `Phone-Number` will be stored.

A collection is created through the button [button] and specified through the *Collection property box*, called up by pressing the Enter key or by double-clicking on the name of the collection (Figure 5.14). It allows us to specify the name, short name, semantic and technical (see below) descriptions, and the list of the entity types (or *tables*) whose entities (*rows*) are to be stored in the collection.

4. ... whose description falls out of the scope of this tutorial. It suffices to know that standard indexes, i.e., those which are not based on hashing techniques, most often are B-tree indexes.

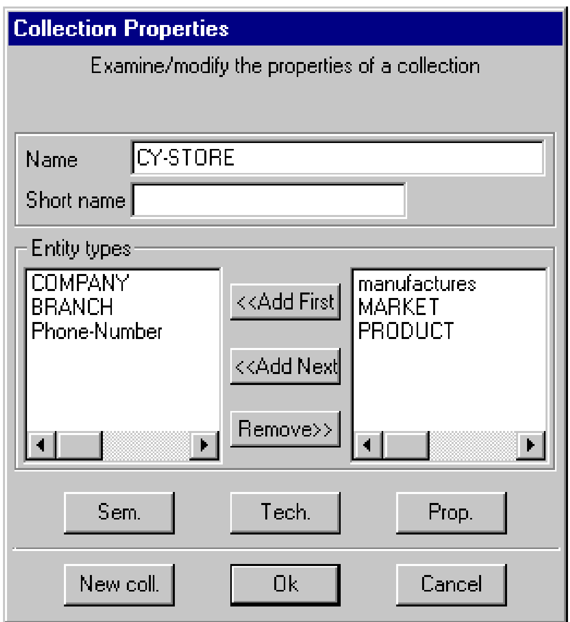


Figure 5.14 - Defining the entity collection CY_STORE.

Any number of entity types can be stored in a collection, and an entity type can be *stored* in any number of collections. However, some DBMS can impose more limited configuration. For instance, many relational DBMS force the rows of each table to be stored in one table space only, though the latter can receive rows from several tables.

These collections appear in all schema views (Figure 5.15 and Figure 5.16).

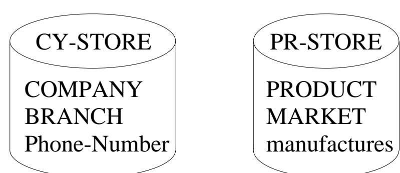


Figure 5.15 - Entity collections: storage units to store table rows in.

```
                    Schema Manufacturing/SQL / Manu

collection CY-STORE
   COMPANY
   BRANCH
   Phone-Number
collection PR-STORE
   PRODUCT
   MARKET
   manufactures

BRANCH
   in CY-STORE
   Com-ID: char (15) [S]
   Name: char (1)
   ...$

COMPANY / COM [S]
   in CY-STORE
   Com-ID: char (15) [S]
   Com-Name: char (25) [S]
   ...

manufactures [S]
   in PR-STORE
   Com-ID: char (15) [S]
   Country: num (3)
   ...
```

Figure 5.16 - Entity collections, according to the Text extended view.

5.9 Name processing

Now we could believe that we are ready to generate the SQL schema that corresponds to the final version of our relational database.

However, a quick look at this schema will show a little but potentially annoying problem: some names include the character "-" (dash), which is invalid in SQL data names. A standard remedy consists in replacing each character "-" by, say, the character "_" (underscore). For instance, `Com-ID` should be replaced by `Com_ID`, and so on.

DB-MAIN has a specific processor for that task. It is called up through **Transform / Name processing**, which opens the *Name Processing panel* (Figure 5.17).

We proceed as follows:

- we set the scope to Global (i.e., processing the whole schema);
- we want to process both the Names and the Short names ...
- ... of the Entity types, Attributes and Collections;
- first, we tell the processor that we want all the names to be converted into uppercase characters (button lower -> uppercase)
- then we define the translation pattern:
    - we click on button Add, therefore opening the New pattern box (Figure 5.18):
    - the character - is typed in the Search for field,
    - the character _ is typed in the Replace by field,
    - and we confirm by clicking on the button OK;
    - the translation pattern "-" -> "_" now appears in the Patterns field (Figure 5.17);

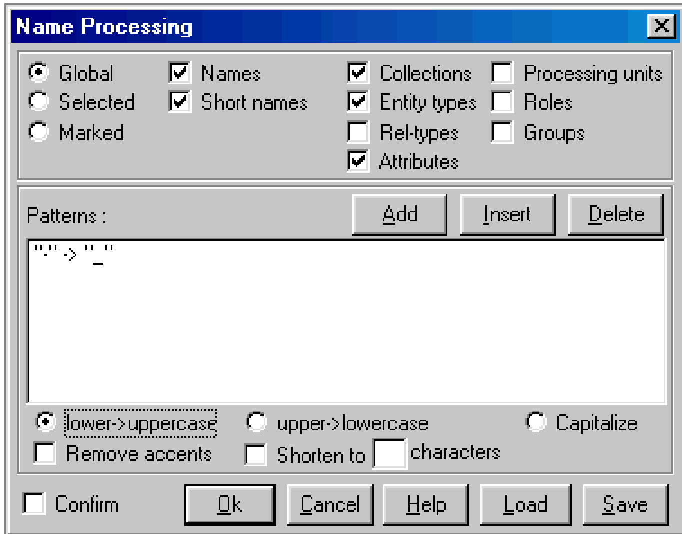


Figure 5.17 - Processing the names of the schema.

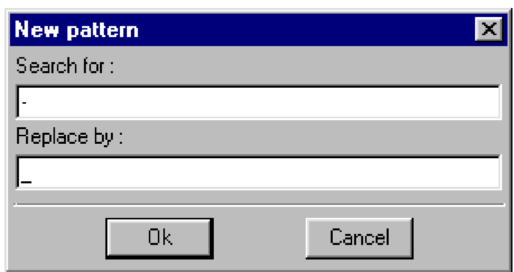


Figure 5.18 - Defining a substitution pattern.

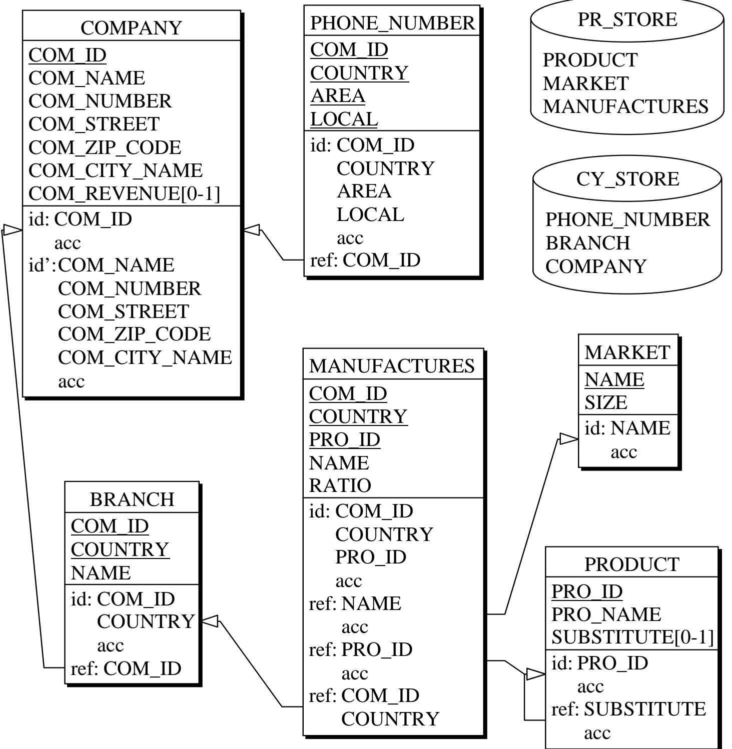


Figure 5.19 - The final physical schema. Note that the prefix access keys have been removed.

- we leave the button Confirm unchecked to avoid being asked for confirmation before each substitution;
- we validate by clicking on the button OK.

All the characters "-" have been replaced with the character "_", just as we wanted them to be and all the names are now in uppercase (Figure 5.19).

The same procedure will also be used to remove space characters or to replace the reserved words it may comprise: no user name can belong to a list that includes such words as *create, table, integer, char, date, index, references, unique, check*, etc.

5.10 SQL code generation

Now we can ask for the SQL translation function as proposed in lesson 3 through the command **File** / **Generate** / **Standard SQL(check)**.

```
-- *********************************************
-- * Standard SQL generation                   *
-- *-------------------------------------------*
-- * Generator date:  Mar  8 2000              *
-- * Generation date: Mon Apr 03 21:50:31 2000 *
-- *********************************************

-- Database Section --

create database Manufacturing;

-- DBSpace Section --

create dbspace CY_STORE;
create dbspace PR_STORE;

-- Table Section --

create table BRANCH (
     COM_ID char(15) not null,
     NAME char(20) not null,
     COUNTRY numeric(3) not null,
     primary key (COM_ID,COUNTRY))
     in CY_STORE;
```

```
create table COMPANY (
     COM_ID char(15) not null,
     COM_NAME char(25) not null,
     COM_NUMBER numeric(5) not null,
     COM_STREET char(20) not null,
     COM_ZIP_CODE numeric(7) not null,
     COM_CITY_NAME char(18) not null,
     COM_REVENUE numeric(12),
     primary key (COM_ID),
     unique (COM_NAME,COM_NUMBER,COM_STREET,COM_ZIP_CODE,
             COM_CITY_NAME))
     in CY_STORE;

create table MANUFACTURES (
     COM_ID char(15) not null,
     COUNTRY numeric(3) not null,
     PRO_ID char(8) not null,
     NAME char(24) not null,
     RATIO numeric(4,4) not null,
     primary key (COM_ID,COUNTRY,PRO_ID))
     in PR_STORE;

create table MARKET (
     NAME char(24) not null,
     SIZE numeric(6) not null,
     primary key (NAME))
     in PR_STORE;

create table PHONE_NUMBER (
     COM_ID char(15) not null,
     LOCAL numeric(8) not null,
     AREA numeric(3) not null,
     COUNTRY numeric(3) not null,
     primary key (COM_ID, LOCAL, AREA, COUNTRY))
     in CY_STORE;

create table PRODUCT (
     PRO_ID char(8) not null,
     PRO_NAME char(25) not null,
     SUBSTITUTE char(8),
     primary key (PRO_ID))
     in PR_STORE;
```

```
-- Constraints Section
-- ___________________

alter table BRANCH add constraint FKBELONGS
     foreign key (COM_ID) references COMPANY;

alter table COMPANY add constraint
     check(exists(select * from PHONE_NUMBER
                  where PHONE_NUMBER.COM_ID = COM_ID));

alter table MANUFACTURES add constraint FKMAN_MAR
     foreign key (NAME) references MARKET;

alter table MANUFACTURES add constraint FKMAN_PRO
     foreign key (PRO_ID) references PRODUCT;

alter table MANUFACTURES add constraint FKMAN_BRA
     foreign key (COM_ID,COUNTRY) references BRANCH;

alter table PHONE_NUMBER add constraint FKCOM_PHO
     foreign key (COM_ID) references COMPANY;

alter table PRODUCT add constraint FKREPLACES
     foreign key (SUBSTITUTE) references PRODUCT;

-- Index Section --

create unique index IDBRANCH on BRANCH (COM_ID,COUNTRY);

create unique index IDCOMPANY on COMPANY (COM_ID);
create unique index IDCOMPANY on COMPANY (COM_NAME,COM_NUMBER,
       COM_STREET,COM_ZIP_CODE,COM_CITY_NAME);
create unique index MANUFACTURES on MANUFACTURES (COM_ID,
       COUNTRY,PRO_ID);
create index FKMAN_MAR on MANUFACTURES (NAME);
create index FKMAN_PRO on MANUFACTURES (PRO_ID);
create unique index IDMARKET on MARKET (NAME);
create unique index IDPHONE_NUMBER on PHONE_NUMBER (COM_ID,LOCAL,
       AREA,COUNTRY);
create unique index IDPRODUCT on PRODUCT (PRO_ID);
create index FKREPLACES on PRODUCT (SUBSTITUTE);
```

Figure 5.20 - The SQL program creating the database.

As we have already mentioned in the first lessons, this SQL text is not quite consistent with the relational schema which it is a translation of. For instance,

the `equ` constraint that appears in the `PHONE_NUMBER` table has been expressed as a mere `ref` constraint. These problems will be addressed in a further lesson.

5.11 Quitting the lesson

We can save the current project and quit DB-MAIN.

Summary of Lesson 5

- In this lesson, we have studied new notions:
- `ref` reference group (or foreign key)
- `equ` reference group
- access key (e.g., index)
- entity collections

- We have also compared *logical schemas* with *conceptual schemas*.

- We have learnt,
- to define a group **New** / **Group**
- to define a reference group
  the `Constraint` button in the Group Property box
- to define an access key
  the `Access key` button in the Group property box
- to define a collection
  New / **Collection**
- to replace substrings in names
  Transform / **Name processing**

Exercises for Lesson 5

5.1 Enter manually[5] a relational logical schema describing the database that was built by the following SQL program:

```
create database RESULTS;

create table STD (
          STD_ID char(15) not null,
          STD_NM char(25) not null,
          STD_PHONE char(10),
          primary key (STD-ID) );

create table LCT (
          LCT_CD char(5) not null,
          LCT_NM char(25) not null,
          primary key (LCT_CD) );

create table CRS (
          CRS_NM char(25) not null,
          LCT_CD char(5) not null,
          HOURS decimal(3) not null,
          primary key (CRS_NM,LCT_CD),
          foreign key (LCT_CD) references LCT) );

create table RES (
          STUD_ID char(15) not null,
          CRS_NM char(25) not null,
          LCT_CD char(5) not null,
          GRADE decimal(5,1),
          primary key (STUD_ID,CRS_NM,LCT_CD),
          foreign key (STUD_ID) reference STUD,
          foreign key (CRS_NM,LCT_CD) references CRS )
);
```

5. Frustratingly (for you!), DB-MAIN includes a powerful tool that can build logical schemas from SQL code. However, using it would make you miss the objective of the exercise.

5.2 This schema is particularly obscure, due to the choice of (too) short names. In fact, the names can be changed to make them more informative. Applying the following substitution leads to a much more readable schema:

```
STD → STUDENT
LCT → LECTURE
CRS → COURSE
RES → RESULT
NM  → NAME
CD  → CODE
```

Use the Name processing function to carry out these replacements. Note that you can add several patterns in the `Patterns` field, so that all the transformations can be executed in one operation.

5.3 Define the access keys (applying **Transform** / **Relational model** will do the job), then generate a new SQL creation program. Though structurally equivalent to the first one, it enjoys a highly desirable quality: readability.

5.4 Try to guess which conceptual schema this logical schema could have come from[6].

5.5 Consider Project `MANU-6` again. Rework the schema hierarchy and some schema constructs in order to propose a neater organization:

- the hierarchy shows the conceptual, logical, physical and coded schemas;
- the physical schema does not include *prefix access keys*.

6. Note that this kind of problem resorts to the Database Reverse Engineering domain, which will be addressed later on.

Lesson 6

Advanced Conceptual Modeling

Objective

This lesson will introduce to more powerful features of the DB-MAIN conceptual model : supertype/subtype relations (is-a), total/partial and exclusive/overlapping subtypes, inheritance, coexistence constraint, exclusive constraint, at-least-one constraint, exactly-one constraint. In addition, a first approach is proposed to schema transformation, and to the reversibility concept.

6.1 Starting Lesson 6

We start DB-MAIN and we create a new project called SEM-6.

6.2 Subtypes and supertypes (is-a relations)

We create a new schema with name ISA and version 1.

Let us suppose that we are describing the activities of *factories* which are in relation to their *suppliers* and their *customers*, which all are *companies*.

In other words, factories, suppliers and customers are companies. In addition, each factory can have customers and can have suppliers. From now on however, we will ignore the latter facts.

If we represent factories, suppliers, customers and companies by entity types FACTORY, SUPPLIER, CUSTOMER and COMPANY respectively, we get the schema of Figure 6.1.

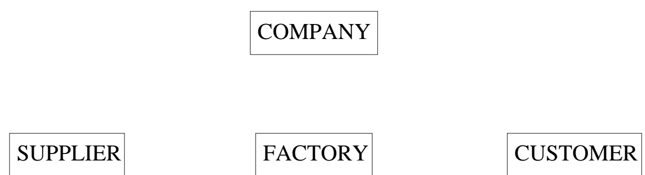


Figure 6.1 - Four unrelated entity types (so far!).

We then have to express some additional facts:

- a factory is a company as well;
- similarly, each supplier is a company;
- and each customer is a company.

Another way to describe these facts is to say that a factory (as well as a supplier and a customer) is a *special kind of* company. This translates in the Entity-relationship model as follows:

- FACTORY is declared a **subtype** of COMPANY;
- SUPPLIER is a **subtype** of COMPANY;
- CUSTOMER is a **subtype** of COMPANY.

Conversely, we can say that COMPANY is a **supertype** of FACTORY, SUPPLIER and CUSTOMER.

To define this *subtype/supertype* relation, we open the Entity box of FACTORY (double-click as usual), and we move the name COMPANY from the right list to the Supertypes list on the left (Figure 6.2).

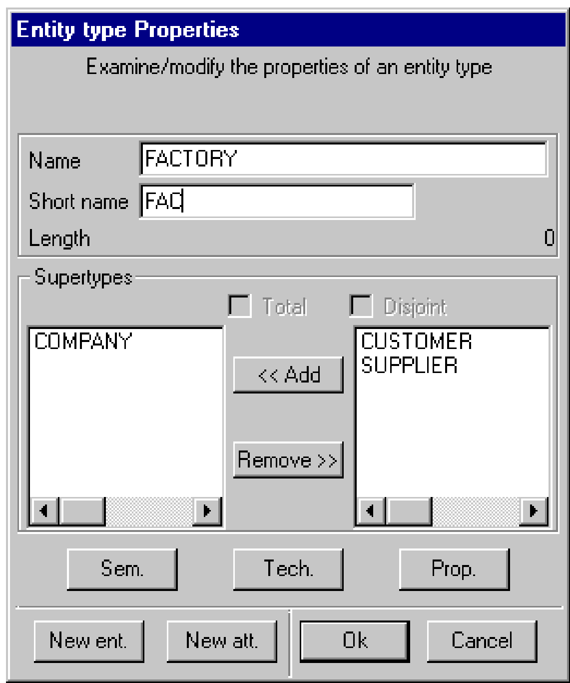


Figure 6.2 - FACTORY is being declared a subtype of COMPANY.

Defining similarly that SUPPLIER and CUSTOMER both have COMPANY as their supertype leads to the schema of Figure 6.3.

It is common to talk about **IS-A relation** between the supertype and its subtypes. The origin of this name lies in the natural language interpretation of the facts modeled in this way:

*each supplier **is a** company, each factory **is a** company, etc*.

The standard view is shown in Figure 6.4 and the extended view in Figure 6.5.

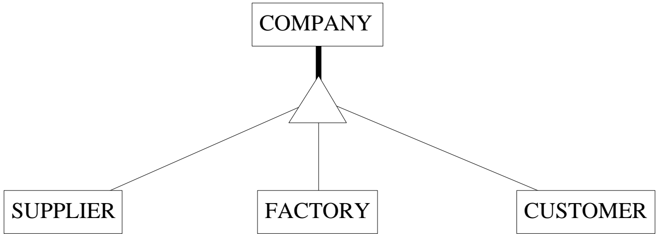


Figure 6.3 - `SUPPLIER`, `FACTORY` and `CUSTOMER` are subtypes of `COMPANY`

```
COMPANY

CUSTOMER
     is-a COMPANY

FACTORY
     is-a COMPANY

etc.
```

Figure 6.4 - The *Text standard view* of IS-A relations.

```
COMPANY / CY
     sub-types: SUPPLIER, CUSTOMER, FACTORY

CUSTOMER / CUS
     is-a COMPANY

FACTORY / FAC
     is-a COMPANY

etc.
```

Figure 6.5 - The *Text extended view* of IS-A relations.

As we can guess by playing with the Entity box, it is possible to state that an entity type has more than one supertype. However, such situations, often called *multiple inheritance*, are much more complicated, and will be ignored in this volume[1].

6.3 Properties of the subtypes of an entity type

So far, we have defined the relation between each subtype and its supertype: each entity of the subtype is an entity of the supertype. So we know that each customer is also a company, and so forth for factories and suppliers.

Now, what about a customer being a supplier as well? ... and about a company which is neither a customer, a factory, nor a supplier?

These questions address two main properties that concern the entity types involved into a supertype/subtype relation. The questions can be stated more formally:

- *are any two subtypes disjoint, or can they overlap*[2]? If the subtypes are pairwise disjoint, then any supertype entity belong to at most one of its subtypes; otherwise it can belong to several subtypes. To assert this property, we will say that the subtypes of entity type `COMPANY` are **Disjoint**. Since this property concerns all the subtypes of `COMPANY`, it is considered to be a property of the supertype.
- *must each entity of the supertype belong to a subtype, or can it be in none of them*? If each supertype entity must belong to at least one subtype, we will say that the subtypes of entity type `COMPANY` are **Total**. This too is a property of the supertype.

When the collection of the subtypes of `E` is both disjoint and total, this collection forms a **Partition**. In a partition, each `E` entity belongs to exactly one subtype.

To allow us to declare these properties, the *Entity box* of the supertype includes two buttons, named **D**isjoint and **T**otal (Figure 6.6). Each can be checked and unchecked independently. When both are checked, the subtypes form a **P**artition, that is, each `COMPANY` entity is of *exactly one* subtype.

1. So far, there is no agreement on what multiple super-types exactly mean, and how to deal with them when validating, transforming and generating a schema. More on this later on.
2. To be more precise, this question concerns the set of entities of each type.

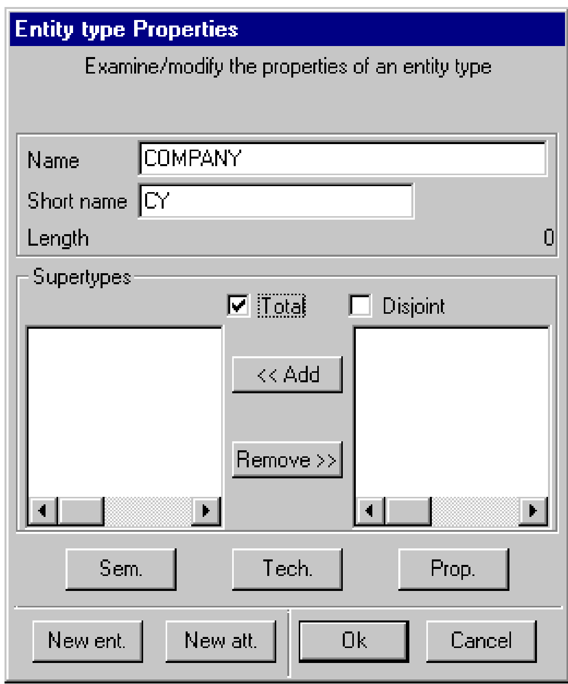


Figure 6.6 - The subtypes of COMPANY **total**ly cover the entity set of COMPANY.

To practice these concepts, we define the subtypes of COMPANY as being **total**:

- we open the *Entity box* of COMPANY (by double-clicking on its name);
- we click on Total;
- we click on OK.

The schema appears as in Figure 6.7 and Figure 6.8.

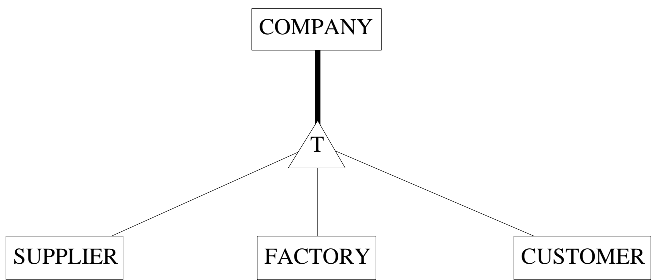


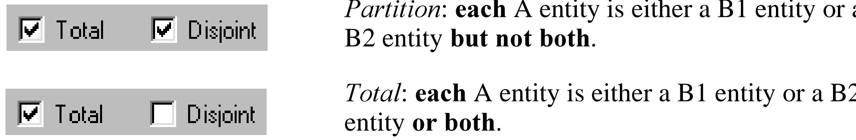
Figure 6.7 - Each `COMPANY` entity also is a `SUPPLIER`, a `FACTORY` or a `CUSTOMER` entity (or several of them).

```
COMPANY / CY
     sub-types (T): SUPPLIER, CUSTOMER, FACTORY

CUSTOMER / CUS
     is-a COMPANY

etc.
```

Figure 6.8 - The Text extended view of the IS-A relations of Figure 6.7.

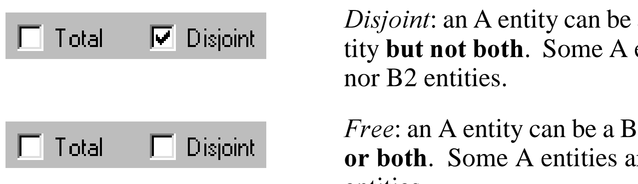
The triangle symbol represents a collection of subtypes. This symbol can include an additional character: **T** for Total, **D** for Disjoint and **P** for Partition. The absence of character means both non-disjoint and non-total, i.e., an overlapping and partial collection of subtypes.

This point being very important in modeling, we will synthesize the different situations in Figure 6.9. It shows a simple IS-A hierarchy made up of supertype A and subtypes B1 and B2. Each pattern is defined as follows.



Partition: **each** A entity is either a B1 entity or a B2 entity **but not both**.

Total: **each** A entity is either a B1 entity or a B2 entity **or both**.



Disjoint: an A entity can be a B1 entity or a B2 entity **but not both**. Some A entities are neither B1 nor B2 entities.

Free: an A entity can be a B1 entity or a B2 entity **or both**. Some A entities are neither B1 nor B2 entities.

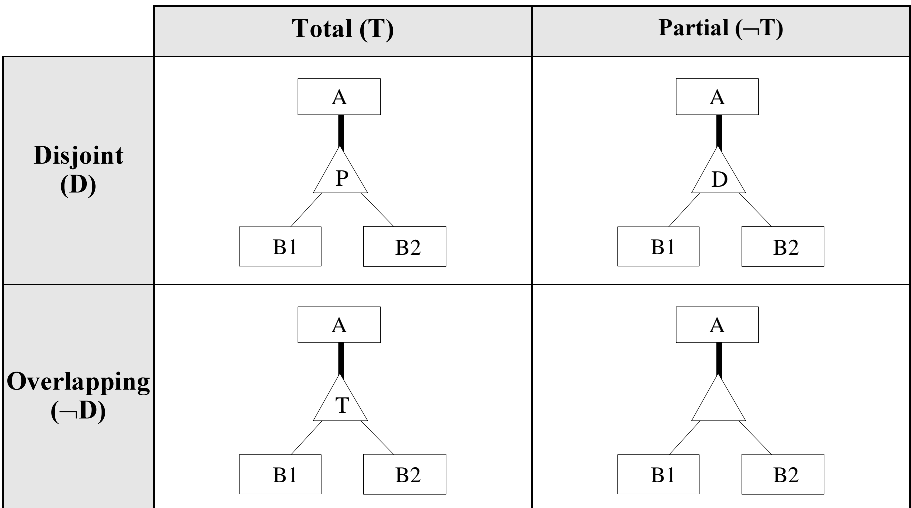


Figure 6.9 - Synthesis of subtype properties.

6.4 Supertype / subtype inheritance

The Supertype/subtype IS-A relation is not as simple as it appears at first glance. One of its most dramatic consequences is the so-called **inheritance** mechanism. To describe it, we need first to enrich our schema a little bit by giving entity types some attributes. Let us record the following facts:

- each company has a name (identifier) and an address;
- each supplier has an account number;
- each factory has a production type;
- each customer has a customer number (identifier), a status and an amount due.

The current schema can be completed easily (Figure 6.10).

Though it is quite correct, this schema does not show explicitly all its contents. For instance, each *customer,* being a *company*, has also a *name* (which identifies it) and an *address*.

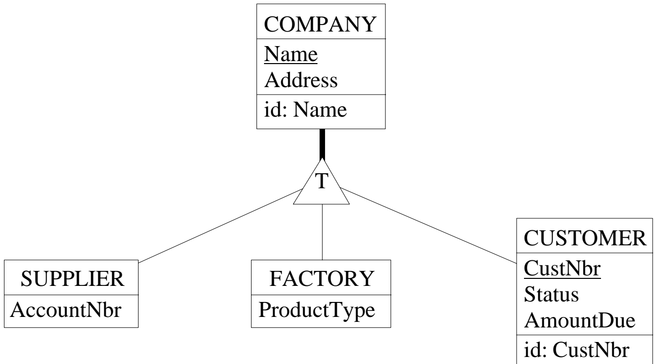


Figure 6.10 - An IS-A hierarchy with attributes.

Thus, the whole list of attributes of entity type CUSTOMER is in fact made of: CustNbr, Name, Address, Status and AmountDue. Among them, CustNbr, Status and AmountDue are called the **proper attributes**, while Name and Address are the **inherited attributes**. In addition, CUSTOMER has two identifiers, namely CustNbr (a proper identifier) and Name (an inherited identifier).

Should the schema show all the attributes and all the identifiers of each entity type, it would appear as in Figure 6.11.

The first version is more concise, while the latter is more informative and includes redundant specifications[3]. However, both views have the same information contents. The only difference is how we have to interpret them.

The concept of inheritance also applies to all the structural properties of the entity types, and is not restricted to attributes and identifiers as discussed so far. More specifically, the subtypes also inherit all the *roles* and the *integrity constraints* of their supertype.

3. For instance, it tells us *twice* that a *customer has a name*: once through an inherited attribute and once as a proper attribute of the supertype.

For instance, if COMPANY is linked to entity type REGION, then CUSTOMER, FACTORY and SUPPLIER are linked to REGION as well (Figure 6.12). Its explicit semantic contents are shown in Figure 6.13.

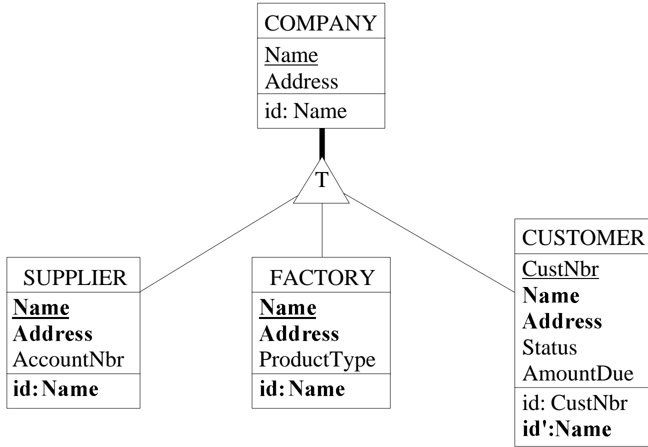


Figure 6.11 - Attribute and identifier inheritance explicitly shown. The inherited components are marked for readability.

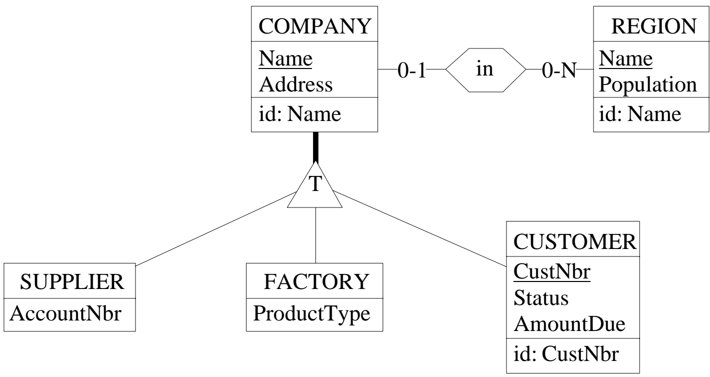


Figure 6.12 - The supertype plays a role in a rel-type.

By comparing both views, the gain of conciseness induced by the supertype/subtype relation is obvious, especially in large schemas. There are other ad-

vantages as well. For instance, inherited components are described only once at the supertype level. Therefore, changing the definition of an attribute (or a role), adding an attribute or deleting an existing attribute, must be done only once. All these changes are automatically applicable to all the subtypes of the supertype.

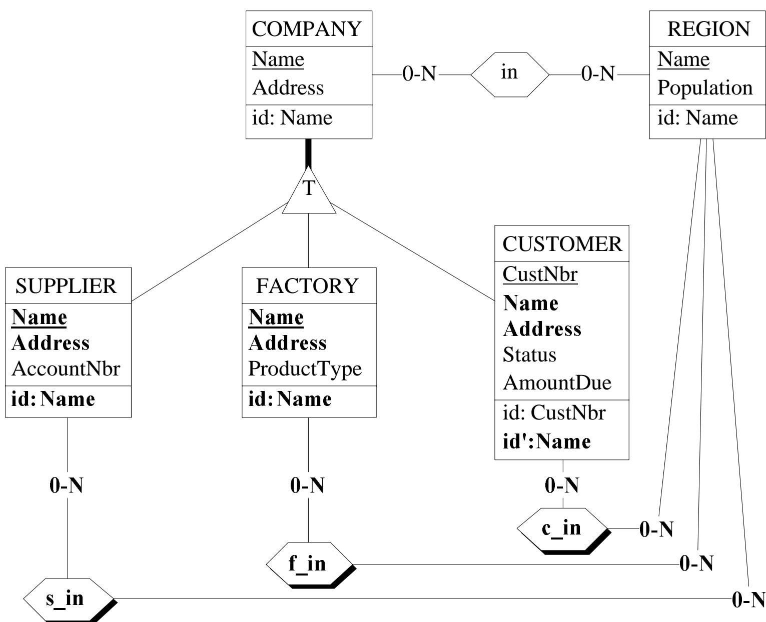


Figure 6.13 - Attribute, identifier and role inheritance shown explicitly.

The drawback of IS-A constructs is that the schema can be less readable. Indeed, the actual attributes (and other components) comprise the proper attributes + the inherited ones.

6.5 Coexistent components of an entity type

We create a new schema, called `Coexistence`, in which we will describe persons who may work in companies and who may be married (a fairly common combination). More precisely, each person is described by its personal

number, its name, the name of his/her spouse, the date s/he was married, the company s/he works for, and the date s/he was hired by this company.

However, not all the persons are married and/or work in a company. Therefore, attributes `SpouseName`, `DateMarried` and `DateHired` are optional and role `works-in.PERSON` is optional too. The corresponding schema looks like Figure 6.14.

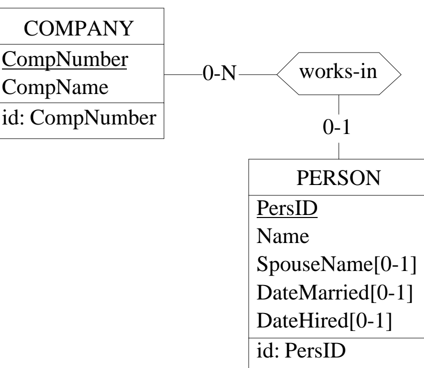


Figure 6.14 - A schema describing persons working in companies.

However, things are not so simple. For instance, all *married persons* have both valid *date married* and valid *spouse name* properties, while *non-married persons* have neither of them.

Similarly, *working persons* have a *date hired* property and a *company* they work in, while *non-working persons* have neither.

We can say that attributes `DateMarried` and `SpouseName` are **coexistent**, i.e., some entities have a value for these attributes, while all the others have no values for them.

DB-MAIN provides us with a specific feature to declare this coexistence constraint: the **coexistence** group. It works as follows:

- we create a group[4] comprising attributes `SpouseName` and `DateMarried`, and we give it the *coexistence* characteristics by clicking on the Coexistence button in the Group box (Figure 6.15);

4. Proceed as usual: select all the components then click on button GR in the Standard tools bar. To open a selected group, just press the Enter key.

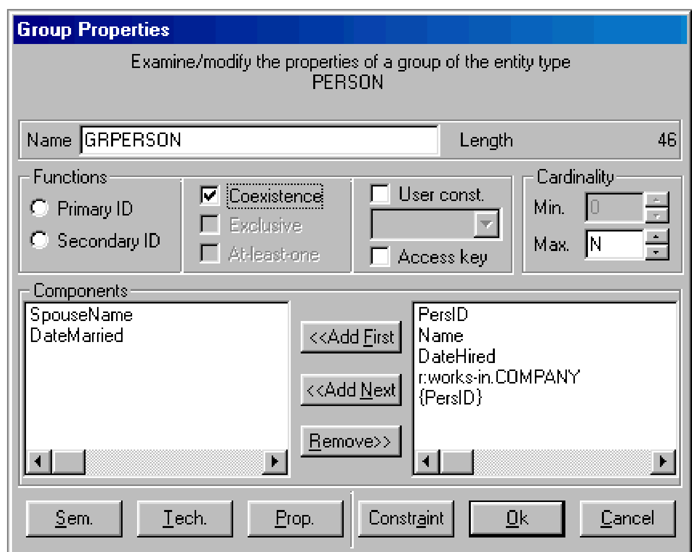


Figure 6.15 - Defining a coexistence group.

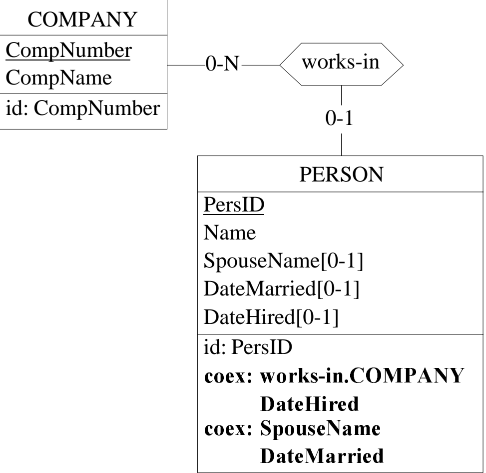


Figure 6.16 - Any person who works in a company must have a date hired, and conversely. All married persons, and only they, have a spouse name and a date of marriage.

- similarly, we define `works-in.COMPANY` and `DateHired` as another coexistence group.

The completed schema is shown in Figure 6.16 and in Figure 6.17.

```
COMPANY
   CompNumber
   CompName
   id: CompNumber
PERSON
   PersID
   Name
   SpouseName[0-1]
   DateMarried[0-1]
   DateHired[0-1]
   id: PersID
  coexist: works-in.COMPANY, DateHired
  coexist: SpouseName, DateMarried

works-in (
    [0-N]: COMPANY
    [0-1]: PERSON )
```

Figure 6.17 - Text view of coexistence constraints.

Note

1. All the components of a coexistence group must be optional. This condition is easy to check for attributes: their cardinality must be [0-j]. For the role components (e.g., `works-in.COMPANY`), the rule is a bit different: the role specifies a relationship type whose other role must be optional, i.e. it has cardinality [0-1]. This rule can be explained by the following interpretation: *a PERSON optionally* (i.e., [0-1]) *works-in a COMPANY.*
2. A coexistence group can also be defined among the attributes of a relationship type.

6.6 Schema transformations : a first glance

To help understand the concept of coexistence constraint, we will propose an equivalent structure which may be more illustrative of the very nature of this constraint. To do so, we will use for the first time a very powerful component

of the DB-MAIN tool, namely its transformation toolkit. This component will be studied in great detail in future lessons, but the current situation is a good opportunity to experiment one of its simplest tool : **attribute aggregation**.

We consider the schema of Figure 6.16, and we proceed as follows:

- we select, by clicking on it, the group that comprises `SpouseName` and `DateMarried`;
- we execute command **Transform** / **Group** / **Aggregation** (Figure 6.18);
- a new attribute is created; we give it the name `Marriage` (or any other name);

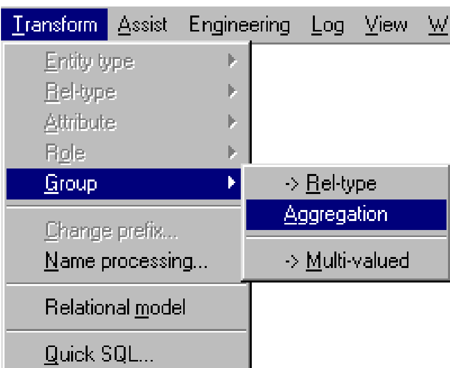


Figure 6.18 - Asking for the aggregation of the components of the selected group into a compound attribute.

As illustrated in Figure 6.19, the set of attributes of `PERSON` has been restructured as follows:

- now, `SpouseName` and `DateMarried` are the components of the new compound attribute `Marriage`;
- these attributes are mandatory for their parent attribute;
- `Marriage` is optional;
- the coexistence constraint has been removed.

It is important to be convinced that the schemas of Figure 6.16 and Figure 6.19 convey exactly the same semantics, i.e., they describe the same portion of the application domain. Indeed, Figure 6.19 tells that a `PERSON` entity can have a `Marriage` value. In this case, it has a value for each of its components, namely `SpouseName` and `DateMarried`. If it has no `Marriage` value,

then, quite naturally, it has no values for the components of this attribute. This is exactly what the coexistence constraint is intended to express.

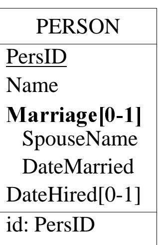


Figure 6.19 - Coexistent group {`SpouseName`,`DateMarriage`} has been transformed into optional compound attribute `Marriage`.

To push the experiment a bit further, we select the attribute `Marriage`, and we execute the command **Transform** / **Attribute** / **Disaggregation**.

(Not really) surprisingly, we get the origin schema! We can draw from this two essential conclusions that will be discussed later on:

1. each transformation is the inverse of the other one: each one erases the effect of the other one; they are called *inverse transformations*;
2. both schemas are equivalent, i.e., they represent exactly the same reality, though through different structures. The choice of one of them will be guided by criteria which are beyond the scope of this lesson. A transformation which replaces a schema with an equivalent one is called *reversible*, or *semantics-preserving*.

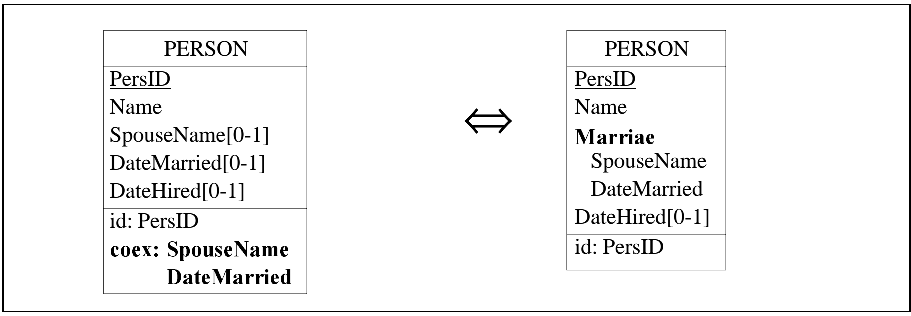


Figure 6.20 - A couple of reversible transformations: Group/Aggregate (left to right) and Attribute/Disaggregate (right to left).

As we will see later on, such a transformation can be summarized as in Figure 6.20.

DB-MAIN offers a fairly large number of schema restructuring techniques, or schema transformations. These are among the most simple, but not the least useful, as will be illustrated in further lessons.

Note

The other *coexistence* group can be processed in a similar way. However, it would need a more sophisticated transformation since it includes attributes **and roles**. Thus, we will leave it to a further lesson.

6.7 Exclusive components of an entity type

This concept is quite similar to the coexistence of components.

Let us record in the current schema information about the *wages* of the persons. Considering that some persons are paid on an hourly basis, while the others are paid at the end of each month, we can define two attributes, namely `HourlyWages` and `MonthlyWages`.

However, no `PERSON` entity can have a value for both attributes. We consider these attributes as *exclusive*.

It is fairly easy to define an **exclusive constraint** in DB-MAIN through an *exclusive group*:

1. we create a new group[5] comprising attributes `HourlyWages` and `MonthlyWages`,
2. we give it the exclusive characteristic by clicking on the Exclusive button in the Group box.

The schema appears as in Figure 6.21.

Let us now consider an additional rule, stating that *companies do not hire married persons*[6]. In other words, a person is married, or works in a company, (or none), but not both.

---

5. Provided no such group already exists. In such a case, just double-click on it and proceed as told in step 2.
6. Non-*equal-opportunity* companies must be modeled as well. Whether describing politically incorrect situations is politically correct or not is beyond the scope of this introduction.

PERSON
<u>PersID</u> Name SpouseName[0-1] DateMarried[0-1] DateHired[0-1] MonthlyWages[0-1] HourlyWages[0-1]
id: PersID coex: works-in.COMPANY DateHired coex: SpouseName DateMarried **excl: MonthlyWages** **HourlyWages**

Figure 6.21 - A person paid monthly cannot be paid per hour, and conversely.

The information concerning the marriage is gathered into a coexistence group {`SpouseName`, `DateMarried`} while the information related to the professional activity of the person is represented by the coexistence group {`works-in.COMPANY`, `DateHired`}.

The **exclusive constraint** is defined by an *exclusive group* as follows:

1. we declare a new group comprising group {`works-in.COMPANY`, `DateHired`} and group {`SpouseName`, `DateMarried`}[7],
2. we give it the *exclusive* characteristic by clicking on the Exclusive button in the Group box.

We get the schema of Figure 6.22.

Notes

1. All the components of an exclusive group must be optional.
2. An exclusive group can also be defined among the attributes of a relationship type.
3. The exclusive constraint can be expressed in a simpler way: `excl: works-in.COMPANY, SpouseName`. Can you explain why this cons-

7. Same procedure as for attributes: select the groups then click on button GR in the Standard tools bar.

traint is equivalent to the former expression?

```
PERSON
------------------------------
PersID
Name
SpouseName[0-1]
DateMarried[0-1]
DateHired[0-1]
MonthlyWages[0-1]
HourlyWages[0-1]
------------------------------
id: PersID
coex: works-in.COMPANY
      DateHired
coex: SpouseName
      DateMarried
excl: MonthlyWages
      HourlyWages
excl: {works-in.COMPANY
      DateHired}
      {SpouseName
      DateMarried}
```

Figure 6.22 - Married persons cannot work in a company, and conversely. A simplified expression will be discussed in the following.

6.8 Groups with *at least one*, or *exactly one*, existing component

Let us consider again the last schema. For the purpose of the demonstration, we delete exclusive group {`MonthlyWages`, `HourlyWages`}.

Now we consider that all the persons are paid, in a way or in another. In our schema, this rule translates as follows: *at least one* of the attributes `HourlyWages` and `MonthlyWages` must have a value.

This property is called the **at-least-one constraint**, and can be specified through an *at-least-one group* as follows:

1. we declare a new group {`Monthly-Wages,Hourly-Wages`},
2. we click on button At-least-one in the Group box.

Without surprise, we get the schema of Figure 6.23.

PERSON
PersID Name SpouseName[0-1] DateMarried[0-1] DateHired[0-1] MonthlyWages[0-1] HourlyWages[0-1]
id: PersID **at-lst-1: MonthlyWages** **HourlyWages**

Figure 6.23 - Every person must be paid, in whatever way(s)!

Very often, such a group will also be given the *exclusive* property, to declare that *one and only one component* must have a value. To state this, we open the group again and we click on the Exclusive button, so that both Exclusive and At-least-1 buttons are checked.

This condition is defined by the **Exactly-one** property (symbolized with **exact-1** in the schema) as shown in Figure 6.24.

PERSON
PersID Name SpouseName[0-1] DateMarried[0-1] DateHired[0-1] MonthlyWages[0-1] HourlyWages[0-1]
id: PersID **exact-1: MonthlyWages** **HourlyWages**

Figure 6.24 - Every person must be paid, but in one way only.

Notes

1. All the components of an *At-least-one* group must be optional.
2. A group cannot have both *Coexistence* and *At-least-one* properties.
3. An *At-least-one* group can also be defined among the attributes of a rela-

tionship type.

6.9 Quitting the lesson

We save the current project and we quit DB-MAIN.

Summary of Lesson 6

- In this lesson, we have studied new notions:
- supertypes, subtypes, supertype/subtype relation
- *total, disjoint and partition* properties
- inheritance
- *coexistence* constraint
- *exclusive* constraint
- *at-least-one* constraint
- *exactly-one* constraint
- schema transformation, inverse transformation, reversible transformation
- the transformation toolkit of DB-MAIN

- We have also learnt:
  - to specify the supertype of an entity type

    in the Entity type box of the subtype : include the name of the supertype in the Supertype list box
  - to define the total, disjoint properties

    in the Entity type box of the supertype : click on the `Total`, `Disjoint` button
  - to define coexistent, exclusive, at-least-one groups

    in the Group box : click on the `Coexistent`, `Exclusive`, `At-least-one` button
  - to define a compound attribute from its components

    if needed, make a group with the components; then :

    Transform / **Group** / **Aggregation**
  - to disaggregate a compound attribute

    Transform / **Attribute** / **Disaggregation**

Exercises for Lesson 6

6.1 In the begining of this lesson, we wrote : ... *factories, suppliers and customers are companies. In addition, each factory can have customers and can have suppliers*. ...

Complete the corresponding schema in order to include these specifications.

6.2 In the same schema, describe the fact that each company can be a *subsidiary* of another company (hint : use a cyclic relationship type). Show how this fact must be interpreted as far as the subtypes are concerned. In other words, make explicit the inherited relationship type. On the basis of this small example, what do you think of the conciseness of the is-a relation ?

6.3 Build a schema (called PERSONAL) representing the following application domain :

The company has employees. Each of them is identified by an employee id, and has a name and an address. An employee can have a personal file. This file has an identifying code, a date and a content. Among the employees, there are clerks and workers. Workers are characterized by a salary, and must be affiliated to a trade union. A clerk has a level and a function. A trade union has a name and an address.

Consider four different hypotheses :

- each employee is either a clerk or a worker, but not both (version 1);
- an employee can be a clerk or a worker, but not both (version 2);
- each employee is either a clerk or a worker, or both (version 3);
- an employee can be a clerk or a worker, or both (version 4).

6.4 Derive from these schemas other schemas (versions 1-1, 2-1, etc) which make explicit all the properties of each entity type by showing the effect of the inheritance mechanism.

6.5 Let us consider the schemas `PERSONNEL` (versions 1, 2, 3, 4). For each of them, derive another schema (versions 1-2, 2-2, etc) in which the *is-a* relation has been eliminated. Proceed as follows:

replace each supertype/subtype relation by a one-to-one relationship type;

Take special care to all the derived integrity constraints.

6.6 Let us consider the schemas PERSONNEL (versions 1, 2, 3, 4). For each of them, derive another schema (versions 1-3, 2-3, etc) in which the is-a relation has been eliminated. Proceed as follows :

- propagate (by inheritance) all the properties of the supertype (attributes, roles, constraints) to each of its subtype;
- remove the supertype.

Pay special attention to all the derived integrity constraints. Be aware that employees who are neither clerks nor workers must be represented anyway.

6.7 Let us consider the schemas PERSONNEL (versions 1, 2, 3, 4). For each of them, derive another schema (versions 1-4, 2-4, etc) in which the is-a relation has been eliminated. Proceed as follows :

- move all the properties of the subtypes to their supertype; for instance, the fact that all clerks have a function can be represented by an optional attribute of EMPLOYEE;
- when all the properties have been pushed up to the supertype, remove the subtypes.

Take a special care to all the derived integrity constraints. The role of an employee (clerk, worker, both or none) should be represented, e.g., through the new attribute Employee-type.

6.8 Can you put forward an opinion concerning these three techniques to eliminate super-type/subtype relations? Some criteria: readability, simplicity, conciseness, complexity of the additional integrity constraints, easiness of translation into a relational database.

Do you think that some of these techniques are more fitted in some situations?

Note.The problem of is-a relation translation is complex, particularly when the database is to be implemented into a standard DBMS (e.g., a relational DBMS). It will be dealt with in a future lesson. Nevertheless, the techniques illustrated in questions 6.5, 6.6 and 6.7 represent the three standard families of representations.

6.9 A relational schema includes two tables, A and B built by the following SQL program (column domains are ignored) :

```
create table A (A1 not null, A2 not null, A3, A4,
           primary key (A1,A2))
create table B (B1 not null, B2, B3, B4,
           primary key (B1),
           foreign key (B3,B4) references A))
```

Represent these structures by a logical schema (as in lesson 5).

Observe that the foreign key is optional. Ideally, two cases only are valid: either both B3 and B4 are null, or both have a value, in which case these values must match an A row. Represent this constraint in the logical schema.

Propose an equivalent conceptual schema.

6.10 Build an entity type PERSON with, a.o., the optional attributes COUNTRY, AREA, LOCAL. Express the fact that these attributes are simultaneously null or valued. Make a compound attribute from them and name it TELEPHONE.

6.11 Add to PERSON the mandatory attribute ADDRESS, made of (NUMBER, STREET and CITY); CITY is in turn a compound attribute comprising ZIP-CODE and CITY-NAME.

- Disaggregate these attributes.
- Make ADDRESS optional then apply the same manipulations.
- Starting from these resulting flat structures, try to go back to the nested structures (*Hint* : if needed define a group [without function] before executing the aggregation of attributes).

6.12 Consider once again the entity type PERSON. Add two entity types, namely COMPANY and ADMINISTRATION. A person can work in a company (where (s)he receives a salary), in an administration (where (s)he has a level) or is unemployed (in which case (s)he receives an unemployment allowance). Add the necessary attributes and/or relationship type to represent these facts. Without resorting to *is-a* relations, add the group constraints expressing the following situations :

- a person must either be in a company, or in an administration or unemployed, but only in one of these situations;

- a person can either be in a company, or in an administration or unemployed, or nothing at all, but only in one of these situations;
- a person must be in a company, or in an administration or unemployed, or in more than one of these situations;
- a person can be in a company, or in an administration or unemployed, in more than one of these situations, or in none of them.

Now, try to express these application domains through is-a relations. What is your opinion when you compare both expressions?

Lesson 7

Conceptual Analysis (1)

Objective

About Lessons 7 to 12
This is the first of a series of lessons dedicated to the analysis and the design of a database. They will describe, through the solving of a representatigve case study, how informal users requirements can be translated into a relational database in a systematic way. While the procedure basically is tool independent, we will see that using a CASE tool such as DB-MAIN can help, even for small projects.

About this lesson
This lesson presents the application domain to be described, and builds the first part of its conceptual schema. It will also introduce the reader to the concept of schema transformation.

7.1 Objective of these lessons

Through the first six lessons, we have mainly discussed the different constructs that make up a database schema, both at the conceptual and logical levels. In addition, we have learned how such schemas can be represented, entered, viewed, manipulated and managed in the DB-MAIN CASE tool. These lessons basically were of a descriptive nature, though the exercises should have given you the opportunity to practice these concepts more actively.

Now it is time to tackle the problems related to the analysis and design of an actual database. Of course, it is out of question to try to address the development of a large scale system, such as those that are developed in companies. Instead, we will propose to build a realistic database related to a part of a small organization, i.e., a small library.

First practice, then theory!

The final objective of this series of lessons is to introduce the novice developer to the principles of database analysis and design. However, for obvious motivation reasons, we have chosen to start with practicing these activities, and to conclude with the systematic description of the methodological principles. Lessons 7 and 8 will be dedicated to conceptual analysis and design of the library database, lesson 9 to 11 will develop its logical design and lesson 12 its physical design. Other volumes will address more sophisticated aspects of databases engineering.

7.2 Conceptual analysis and design

We will follow a simple procedure which applies elegantly to problems for which we are provided with a semi-formal textual description. Such a description consists in factual sentences describing the application domain (i.e., the problem to solve or the system to describe) in terms of its concepts, of their properties and of their organizing rules. In other words, such a text can be interpreted as a linguistic expression of the future conceptual schema.

We will decompose this text, and somewhat rework it, in order to obtain a list of elementary propositions that are easy to interpret and to translate into Entity-relationship constructs.

Of course, this text may include some flaws, such as redundencies and conflicting information, and can lack some important information as well. Consequently, we may be forced to ask people from the application domain for additional information.

You should be aware that, in actual situations, many other sources of information can be used to contribute to the conceptual analysis. Let us mention administrative and legal documents, observation of working procedures, forms and other documents, screen layouts and printed reports, existing files and databases, existing programs. They may need more advanced techniques and methods, and will be addressed in other volumes.

7.3 The case study

The problem that will be solved in the following lessons is to build a database describing a small library organized by a consortium of companies.

A descriptive document is available. This document results from the interview of the employees in charge of the management and storage of the books, advising and helping readers and borrowers, and managing the borrowings of books. We assume that the employees can be contacted if needed.

The text of Figure 7.1 comprises the main excerpts of the interviews.

7.4 The analysis

This text will be decomposed into elementary sentences, each of them stating an elementary fact about the application domain.

Each elementary sentence will be **analyzed** in order to interpret the new facts that it tells us. Then, if needed, **actions** will be taken on the current schema to integrate the knowledge extracted from this sentence.

At starting time, the current schema is empty.

To make the development more readable, we have organized the analysis into specific sections, each dedicated to a major concept of the application domain. This can be perceived as a bit artificial, since some concepts may appear thanks to the contribution of several sentences scattered through the text.

A book is considered a piece of literary, scientific or technical writing. Every book has an identifying number, a title, a publisher, a first published date, key words, and an abstract (the abstracts are being encoded), the names of its authors, and its bibliographic references (i.e., the books it references).

For each book, the library has acquired a certain number (0, 1 or more) of copies. The copies of a given book have distinct serial numbers. For each copy, the date it was acquired is known, as well as its location in the library (i.e., the store, the shelf and the row in which it is normally stored), its borrower (if any), the number of volumes it comprises. It appears that one cannot borrow one individual volume, but that one must borrow all the volumes of a copy. In addition, the copies of a given book may have different numbers of volumes. A book is also characterized by its physical state (new, used, worn, torn, damaged, etc), specified by a one-character code, and by an optional comment on this state.

The author of a book has a name, a first name and an origin (i.e., the organization which (s)he came from when the book was written). For some authors, only the name is known. The employees admit that two authors may have the same name (and first name), but such a situation does not seem to raise any problem. Only the authors of books known by the library are recorded.

A copy can be borrowed, at a given date, by a borrower. Borrowers are identified by a personal id. The library records the name, the first name, the address (name of the company, street, zip-code and city name), as well as the phone numbers of each borrower. In addition, when (s)he is absent, another borrower (who is responsible for the former) can be contacted instead. A copy is borrowed on behalf of a project (identified by its title, but also by its internal code). When a copy is brought back to the desk, the employee records the following information on this copy: borrowing date, current date, borrower and project, then the copy is put in a basket from which it is extracted at the end of the day to be stored in its location, so that it can be available again from the following day on...

Figure 7.1 - The interview report.

7.5 Starting Lesson 7

We execute DB-MAIN and we create a new project called `concep-7`, through command **File** / **New project**.

7.6 Starting the analysis

We create a new schema called `LIBRARY` with a version label `Conceptual`.

For each sentence, we proceed to an analysis phase (**Analysis**) then we update the current schema (**Action**). As already said, we have organized the discussion in homogeneous sections.

7.7 The books

(1) A book is considered a piece of literary, scientific or technical writing.

Analysis: this proposition puts forward the (probably) central concept of book. In addition, the fragment "*is considered a piece of ... writing*" can be interpreted in two ways (Figure 7.2):

1. as the definition of the concept of book;
2. as the specification of three categories of books.

Action: we create a new entity type with name `BOOK`. According to the interpretation chosen, we create, or not, three subtypes, as illustrated below. In the following, we will adopt the simplest version, i.e., the left-hand side one. The sentence *"piece of literary, scientific or technical writing"* is entered in the semantic description of `BOOK`.

We could add an attribute that define the category of the book. However, we will ignore it unless the rest of the text mentions it again.

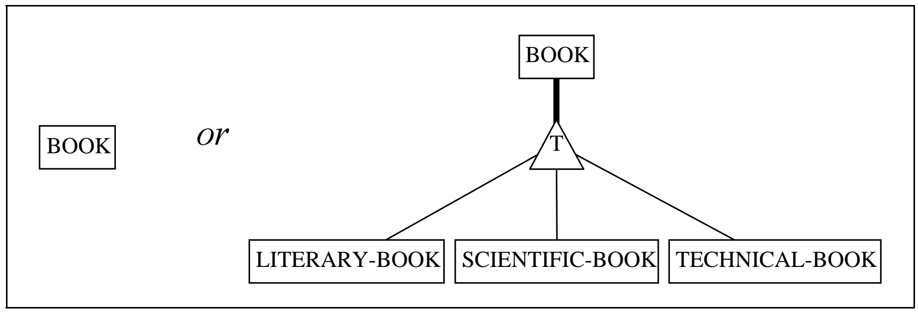


Figure 7.2 - Two views of the concept of book.

(2) Every book has an identifying number,

Analysis: this is a mere property of the books.

Action: we add an attribute called BOOK-ID. We declare it the primary id of BOOK (Figure 7.3).

(3) a title,

Analysis: property of books.

Action: we add the attribute TITLE (Figure 7.3).

(4) a publisher,

Analysis: at first glance, a property of books. However, we could have interpreted this proposition as the existence of a major concept of the application domain.

Action: we choose the simplest interpretation, and we add the attribute PUBLISHER to BOOK. Later on the concept of publisher may appear essential; in this case we will transform this attribute into an entity type (Figure 7.3).

(5) a first published date,

Analysis: obviously a property of books.

Action: we add the attribute DATE-PUBLISHED to BOOK (Figure 7.3).

(6) *key words,*

Analysis: depending on whether bibliographic retrieval is considered important or not, the concept of key word will be perceived as a major one, or as a mere property of books.

Action: without any other information, we choose the minimal interpretation, and we define the attribute `KEY-WORD` of `BOOK`. Should the concept become more important in the future, we will transform this attribute into an entity type.

This attribute is obviously optional and multivalued. By further discussing with the employees, we are told that ten key words is an absolute maximum. Hence the cardinality `[0-10]` (Figure 7.3).

(7) *and an abstract*

Analysis: a property of books.

Action: we add the attribute `ABSTRACT` to `BOOK` (Figure 7.3).

(8) *(the abstracts are being encoded)*

Analysis: the abstract is optional.

Action: the cardinality is set to `[0-1]` (Figure 7.3).

(9) *its author names,*

Analysis: the author names can be understood either as a multiple property of books, or as an important concept of the application domain.

Action: we choose to represent them by the multivalued attribute `AUTHOR` of `BOOK`. The cardinality is undefined, so let us set it to `[0-N]`.

The resulting schema then appears as in Figure 7.3.

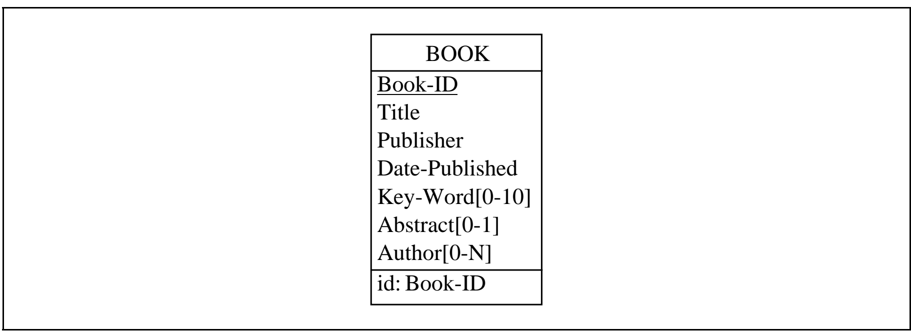


Figure 7.3 - The first attributes of BOOK.

(10) and its bibliographic references (i.e., the books it references).

Analysis: we interpret this proposition as follows: a book can reference an arbitrary number of other books, and any book can be referenced by other books. We can translate this by a cyclic relationship type.

Action: we define the relationship type `BIBLIO-REF`. One of the roles is named `ORIGIN` (the book which includes the reference) while the other one is named `REFERENCE` (the book which appears as a reference in the former one). Both cardinalities are `[0-N]` (Figure 7.4).

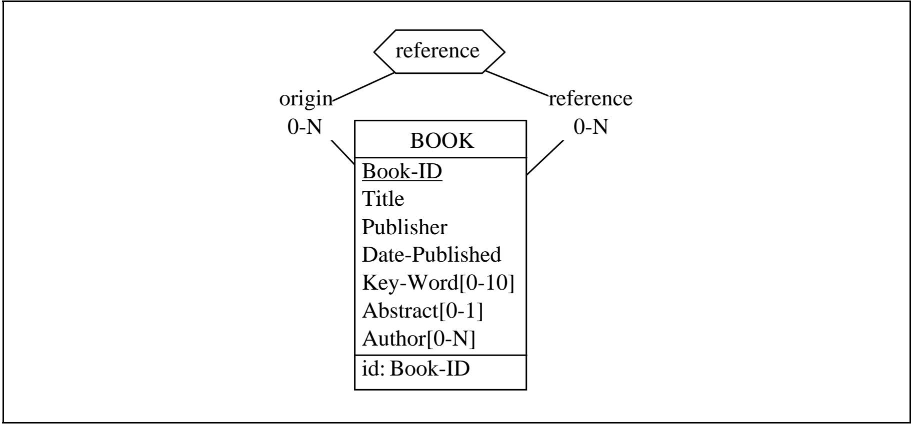


Figure 7.4 - Books can be the origin of references to other books.

7.8 The copies

(11) For each book, the library has acquired a certain number (0, 1 or more) of copies.

Analysis: at this stage, it is still unclear whether the concept of copy is important in the application domain. At least, the number of copies must be known.

Action: wait and see.

(12) The copies of a given book have distinct serial numbers.

Analysis: obviously, more must be recorded about the copies. The list of serial numbers is a good candidate.

Action: we define a new attribute SER-NUMBER of BOOK. Since the exact number of copies is unknown, we set the cardinality to [0-N] (Figure 7.5).

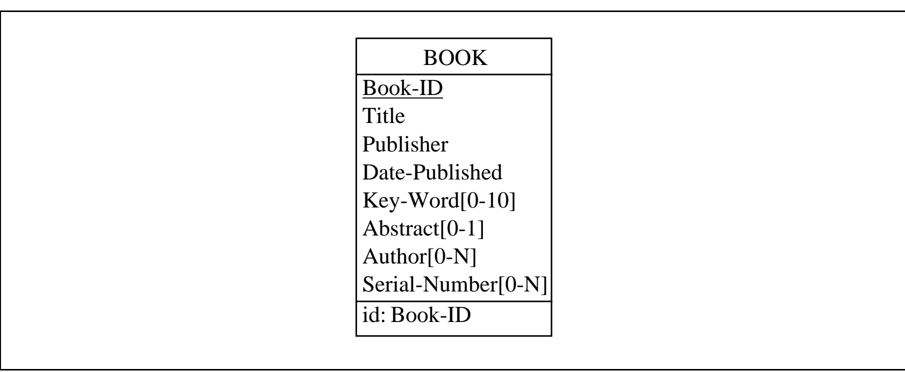


Figure 7.5 - The copies of a book have distinct serial numbers.

(13) For each copy, the date it was acquired is known,

Analysis: now, there is too much information about copies to keep them as a mere property of books. We propose to include the copies among the main concepts of the application domain (not very surprising indeed). The date will become a specific property of this concept.

Action: the first thing to do is to define an entity type representing the copies. There are two possible ways to do so.

The *first* one consists in deleting the attribute SER-NUMBER[0-N] and in creating a new entity type COPY with the attributes SER-NUMBER and DATE-ACQUIRED.

The *second* one is much more elegant, and allows us to illustrate a new way of reasoning when building a conceptual schema incrementally: namely *schema transformation*.

Let us select the attribute SER-NUMBER (by clicking on its name), and let us promote it to the entity type status. We call the function **Transform** / **Attribute** / **-> Entity type**. We choose the *Instance representation* technique, that will be discussed later on. This transformation replaces the selected attribute with an entity type. We choose the name COPY for the entity type, and the name of for the relationship type between BOOK and COPY.

The transformation can be represented as follows:

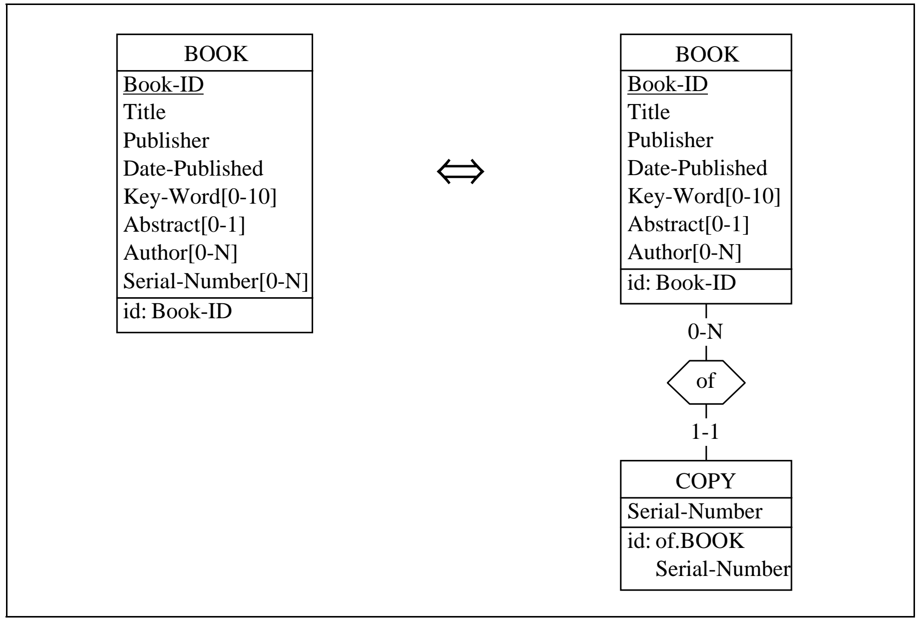


Figure 7.6 - Copies need to be represented by entity types.

Before going on, we have to discuss in further detail the notion of *schema transformation*, which was already introduced in Lesson 6 (Section 6.6).

First of all let us convince ourself that both schemas are strictly equivalent as far as their information contents are concerned. Can we imagine a situation described by one of these schemas which cannot be described by the other one? A book without copies? A book with 419 copies? Several books which have each a copy with sequence number 14? A book with copies identified by numbers 1, 3, 4, 18, 24? Note in particular that none allow two copies of the same book to have the same number. It is possible to prove that these schemas are equivalent in any circumstance, but relying on our intuition will suffice for now[1].

Though this specific transformation will be described in further detail at the end of this lesson (see the *Addendum*), we will give some indications which may be necessary to understand its application in the current situation.

It is intended to replace an attribute with an equivalent entity type. Two techniques are proposed, namely *Instance representation* and *Value representation.*

Through the *Instance representation* technique, each instance of the attribute `SER-NUMBER` is represented by an individual `COPY` entity. For instance, if 2627 books have a copy with number 5, there will be 2627 `COPY` entities with `SER-NUMBER` value 5. Consequently, `SER-NUMBER` is in no way the identifier of `COPY`, but it is a component of its identifier. In addition, the rel-type `of` is *one-to-many*.

With the *Value representation* technique, each distinct value of `SER-NUMBER` is represented by an individual `COPY` entity. For instance, in the situation described above, there will be only 1 `COPY` entity with `SER-NUMBER` value 5. `SER-NUMBER` is the identifier of `COPY`. `of` is now *many-to-many*. Of course, such an entity type has no particular meaning in this situation; therefore, we have chosen the *Instance representation* technique.

Finally, let us observe that there exists another transformation through which we can go back to the initial schema: transforming an entity type (at least some kind of entity types) into an attribute. To experiment this, we select the entity type `COPY` we have just produced and we call the command **Transform / En-**

---

1. This transformation, and some others, have been defined in the reference: Hainaut, J-L, *Entity-generating Transformations for Entity-relationship Schemas*, in Proc. of the 10th Int. Conf. on the ER Approach (San Mateo, 1991), North-Holland, 1992.

tity type / -> Attribute. The final schema is the same as the starting one, which should not be so surprising.

Now we can introduce the date the copy was acquired: we add the attribute `DATE-ACQUIRED` to entity type `COPY` (Figure 7.7).

(14) as well as its location in the library

Analysis: a property of the copies.

Action: add the attribute `LOCATION` to entity type `COPY` (Figure 7.7).

(15) (i.e., the store, the shelf and the row in which it is normally stored),

Analysis: this is a definition of what is the location of the copy.

Action: we make `LOCATION` a compound attribute with components `STORE`, `SHELF` and `ROW` (Figure 7.7).

(16) its borrower (if any),

Analysis: are borrowers an essential concept in this application domain, or are they a property of copies only? Waiting for further information, we decide to represent the borrower of each copy by an attribute. This attribute is optional (some copies only are borrowed at a given time) and single-valued (a copy is borrowed only once at a time).

Action: we add a new attribute (`BORROWER[0-1]`) to entity type `COPY` (Figure 7.7).

(17) the number of volumes it comprises.

Analysis: the problem seems to be the same as for the copies of a book: do we represent the number of volumes (as an integer) or each volume individually (as a multivalued attribute or even as an entity type)? We choose the first representation. A second question: have all the copies of a given book the same number of volumes? If they have, it is best to associate the attribute to `BOOK` instead of to `COPY`. Following the text (but we should be ready to change our mind), we associate the attribute to `COPY`.

Action: we add the attribute `NBR-OF-VOLUMES` to `COPY` (Figure 7.7).

(18) It appears that one cannot borrow one individual volume, but that one must borrow all the volumes of a copy.

Analysis: this rule indicates that there is no need to represent each volume individually.

Action: none.

(19) In addition, the copies of a given book may have different numbers of volumes.

Analysis: this is a confirmation that the number of volumes characterizes the copies and not the books.

Action: none.

(20) A book is also characterized by its physical state (new, used, worn, torn, damaged, etc), specified by a one-character code,

Analysis: apparently a property of books. However, this is not realistic: the physical state is a property of the physical objects, i.e., copies. Some copies of a book can be in good condition, while others can be severely damaged. Therefore, book must be understood as copy instead.

Action: we add the attribute `STATE` to entity type `COPY`. Its domain of value is made of individual characters (Figure 7.7).

(21) and by an optional comment on this state.

Analysis: an optional property of copies.

Action: we add an optional attribute (`STATE-COMMENT`) to `COPY`.

Finally, the schema appears as in Figure 7.7.

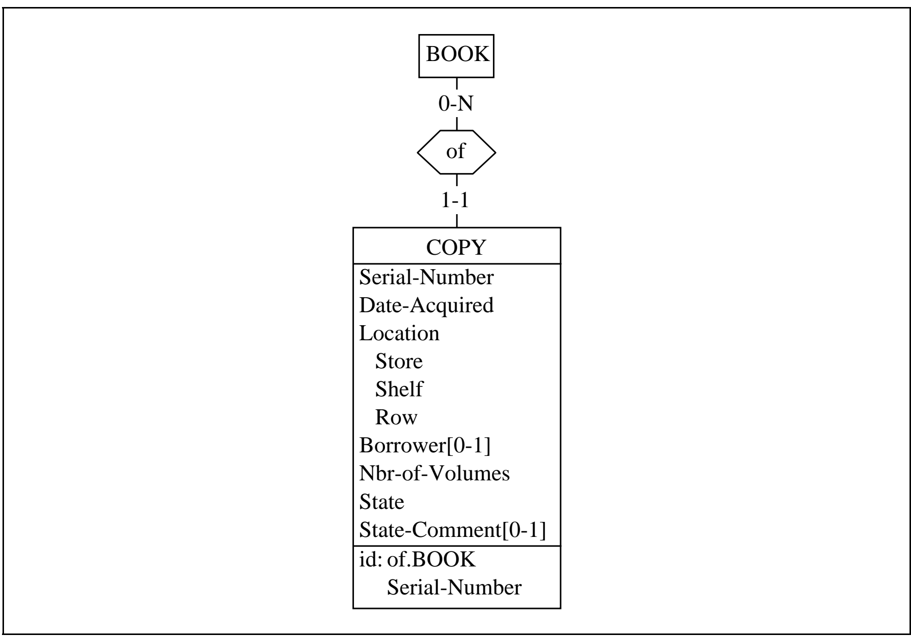


Figure 7.7 - COPY has some new attributes.

7.9 The authors

(22) The author of a book has a name,

Analysis: this does not change our first perception. It just precises the kind of information conveyed by the attribute `Author` of `BOOK`.

Action: we can adjust the type of values of `Author` so that it can represent author names.

(23) a first name,

Analysis: the name of the authors is extended. We could make `Author` a compound attribute (with components `Name` and `First-Name`). We could also propose to represent authors by a specific entity type. We choose the first structure.

Action: we select the attribute Author, and we give it a component through command **New** / **Attribute** / **First att**. In this way, we add a first component to Author, which automatically becomes compound. This new attribute is named Name. Then we create an additional attribute by clicking on the Next att. button in the Attribute box of Name. We call this new attribute First-Name.

The structure of BOOK is shown in (Figure 7.8):

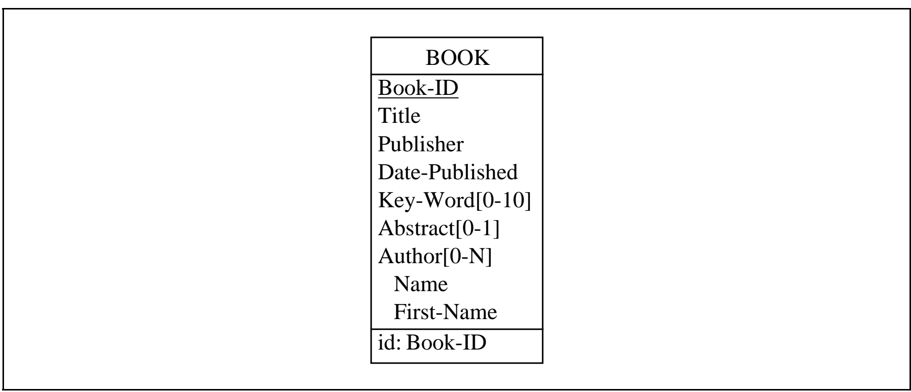


Figure 7.8 - Details are obtained on the authors of books.

(24) and an origin (i.e., the organization which (s)he came from when the book was written).,

Analysis: OK, there is a little too much information for a compound attribute. We must represent authors by an entity type. Then we will represent the origin of the authors.

Action: since the authors already are represented as components of BOOK, we will choose the transformational approach (in the same way as for defining copies from books).

We select the attribute AUTHOR, and we call the command **Transform** / **Attribute** / **-> Entity type**. Do we need to represent each instance of AUTHOR, or each distinct value of AUTHOR? In other words, if "Hugo, Victor" is the author of 6 books (in this library), do we represent him by 6 AUTHOR entities, or by one only? Obviously the answer is one only. It seems best to represent each author by one and

only one AUTHOR entity. Therefore, we must choose the *Value representation* technique. The result is in Figure 7.9.

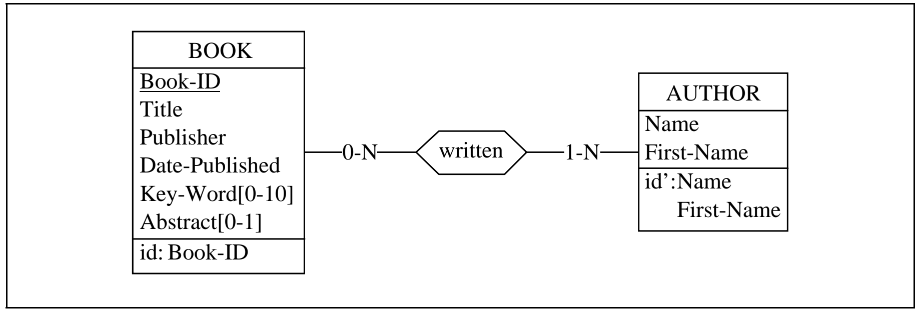


Figure 7.9 - Due to additional information on authors, it is best to represent them by entity type AUTHOR. Technically, the attributes of AUTHOR form its identifier. To be confirmed.

We can then add the attribute `Origin` to `AUTHOR` (Figure 7.10).

(25) For some authors, only the name is known.

Analysis: we are told that `First-Name`, `Birth-Date` and `Origin` are optional attributes.

Action: we set their cardinality to [0-1]. The result is not quite correct, since the primary identifier now include an optional component. To be checked later on (Figure 7.10).

(26) The employees admit that two authors may have the same name (and first name), but such a situation does not seem to raise any problem.

Analysis: this is a conflicting information. To be honest, since this was produced by the transformation, we have taken for granted that (1) all the `Author` values of a `BOOK` entity were distinct, and (2) two authors must have distinct name and first name. Hence the identifier of `AUTHOR`. We learn here that there is no known way to uniquely characterized the authors. We have to remove the primary identifier of `AUTHOR`.

Action: we select the identifier of AUTHOR (by clicking on its declaration), and we delete it by pressing the Delete key (Figure 7.10).

(27) Only the authors of books known by the library are recorded.

Analysis: every author has written at least one book. This confirms the [1-N] cardinality of the role AUTHOR in written.

Action: none.

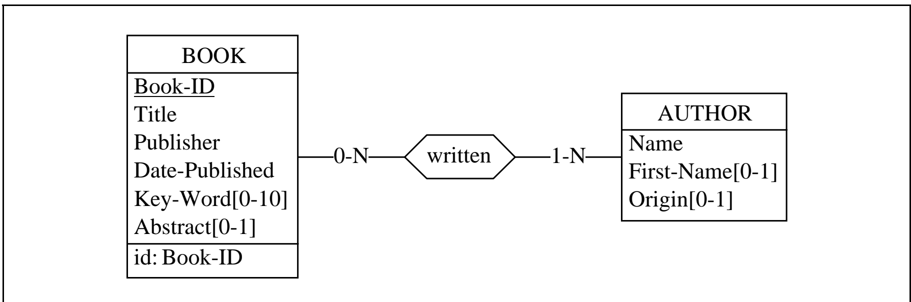


Figure 7.10 - New attributes of AUTHOR. In fact, AUTHOR has no formal identifier.

7.10 The current schema

It is a good time to close this lesson. Of course, the analysis is far from finished, but we will leave its completion to the next lesson. Before quitting, let us just have a look to the current state of the schema (Figure 7.11).

7.11 Quitting the lesson

We save the current project through **File** / **Save project** under the name concep-7 and we quit DB-MAIN.

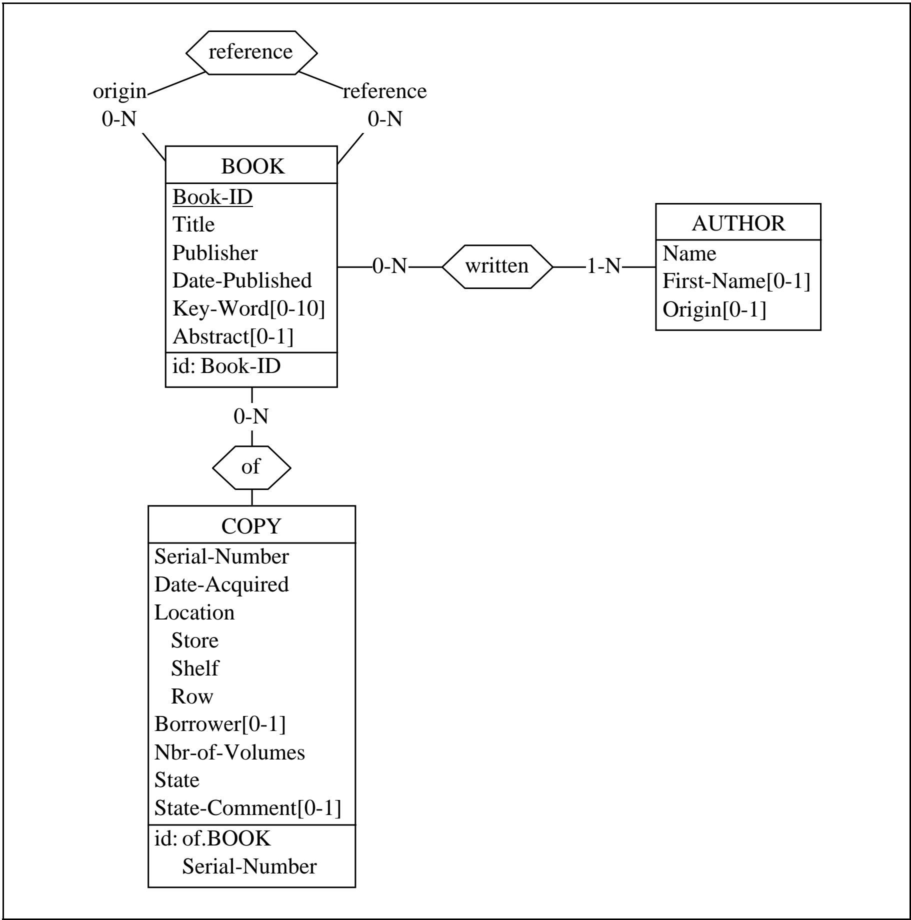


Figure 7.11 - The final schema - First version.

Technical addendum

In this lesson, we have been faced with the problem of replacing an attribute with an entity type, particularly when a concept was represented with an attribute, then happened to have more properties than earlier thought, or should be linked with other concepts.

This replacement operation, called a **transformation**, is an important tool in database engineering. It deserve to be discussed in more technical detail, which is the aim of this addendum.

7.12 The *attribute/entity type* transformation

We will avoid a formal analysis of this technique. Rather, we will present some important examples of application.

The reader is invited to learn from these examples the principles of the main variants of the attribute/entity type transformation. Anyway, the DB-MAIN tool itself will prove the best companion when you want to practice this technique. So, try by yourself on your own examples.

The transformation that will be analyzed has an interesting property: it is **semantics-preserving**, i.e., the resulting schema has exactly (no more, no less) the same information contents as the source schema. One essential consequence of this equivalence is that any semantics-preserving transformation has an inverse with which we could transform the resulting schema into the source schema.

Though the *attribute/entity type* transformation is semantics-preserving, and therefore can be read both way, the following examples have been prepared to be interpreted from left to right. In other words, we present them as *attribute to entity-type* transformations. Reading them from right to left gives interesting hints on how to *reduce an entity type into a mere attribute* without loss of information.

Processing single-valued mandatory attributes

In the first examples, we transform a single-valued, mandatory, attribute (`NAME`) into an entity type. First, we apply the *Value representation* technique. The entity type `NAME` represents a dictionary of all the names corresponding

to at least one person. Each name is represented once and only once. The relationship type `of` is *one-to-many* (Figure 7.12).

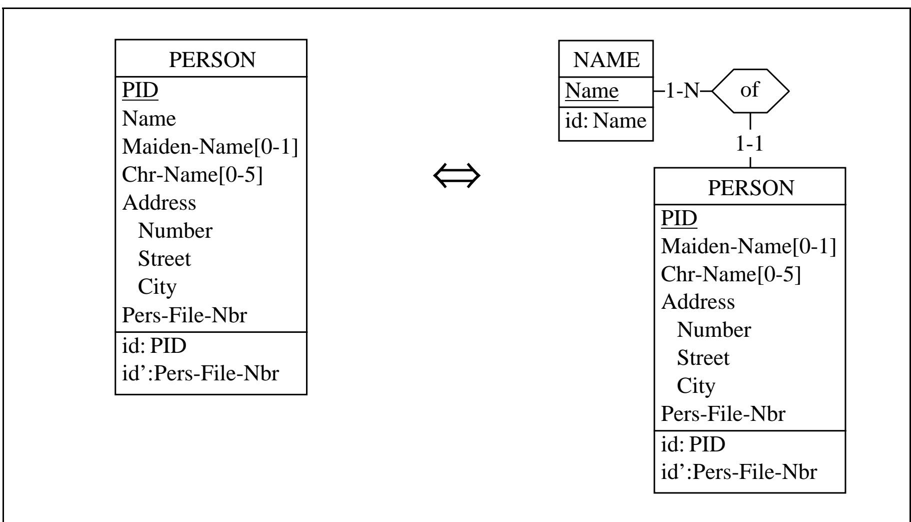


Figure 7.12 - Extracting attribute Name as an entity type through *Value representation*.

Then, we apply the *Instance representation* technique. Each `NAME-OF-PERS` entity represents the `NAME` instance of one `PERSON` entity. Each name can be represented more than once. The relationship type `of` is *one-to-one*, and `NAME-PERS` has no identifier (in fact, `PERSON` is an implicit identifier of it, since a person has only one `NAME` entity) (Figure 7.13).

Processing single-valued optional attributes

In the following example, we transform a single-valued, optional, attribute (`MAIDEN-NAME`) into an entity type. We apply the *Value representation* technique; experiment the other one by yourself. The discussion is the same as for `NAME`, except that the attribute `MAIDEN-NAME` (left), and the relationship type `of` (right) both are optional (Figure 7.14).

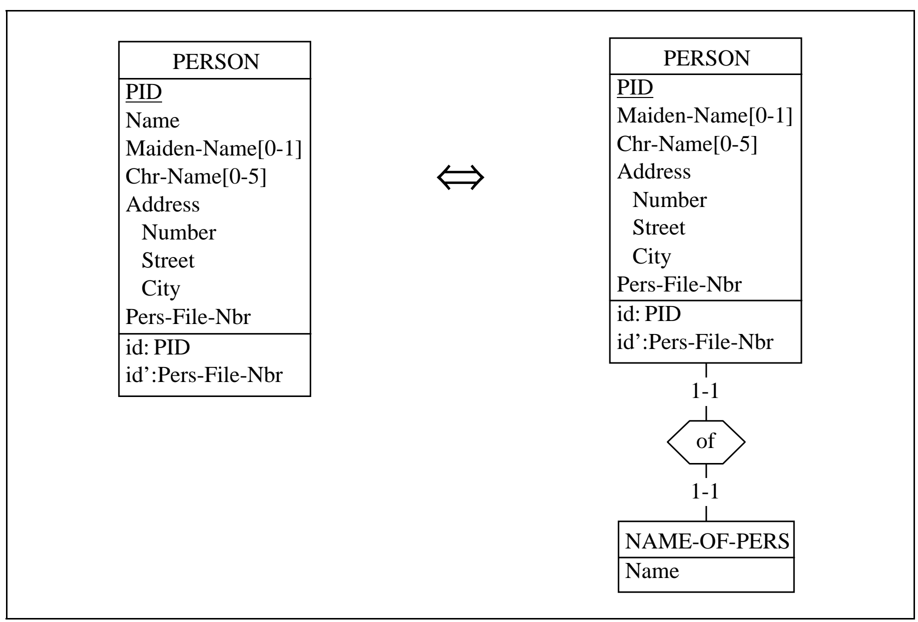


Figure 7.13 - Extracting attribute Name as an entity type through *Instance representation*.

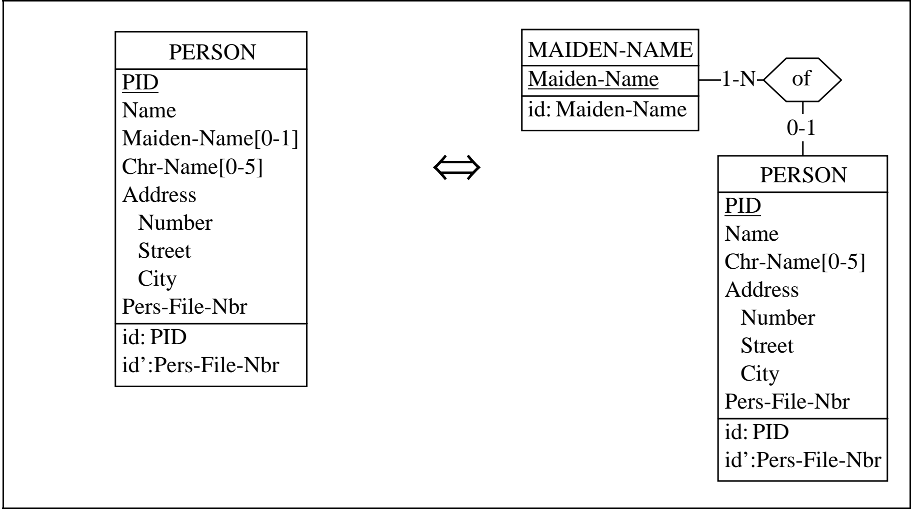


Figure 7.14 - Extracting attribute Maiden-Name as an entity type through *Value representation*.

Processing multivalued attributes

Let us now process a multivalued attribute (CHR-NAME[0-5]). First, we apply the *Value representation* technique. The entity type CHR-NAME represents a dictionary of all the christian names corresponding to at least one person. Each christian name is represented once and only once. The relationship type of is *many-to-many* (Figure 7.15).

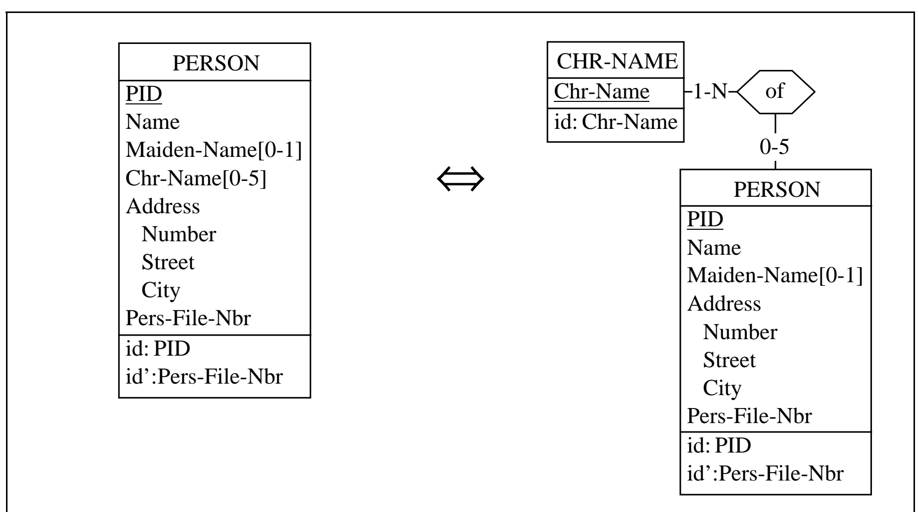


Figure 7.15 - Extracting attribute Chr-Name as an entity type through *Value representation*.

Then we apply the *Instance representation* technique. The entity type CHR-NAME-OF-PERS does not represent a dictionary, but collects all the CHR-NAME values of all the PERSON entities. There are as many CHR-NAME-OF-PERS with value "*Marie*" as there are persons with this christian name. The relationship type of is *one-to-many* (Figure 7.16).

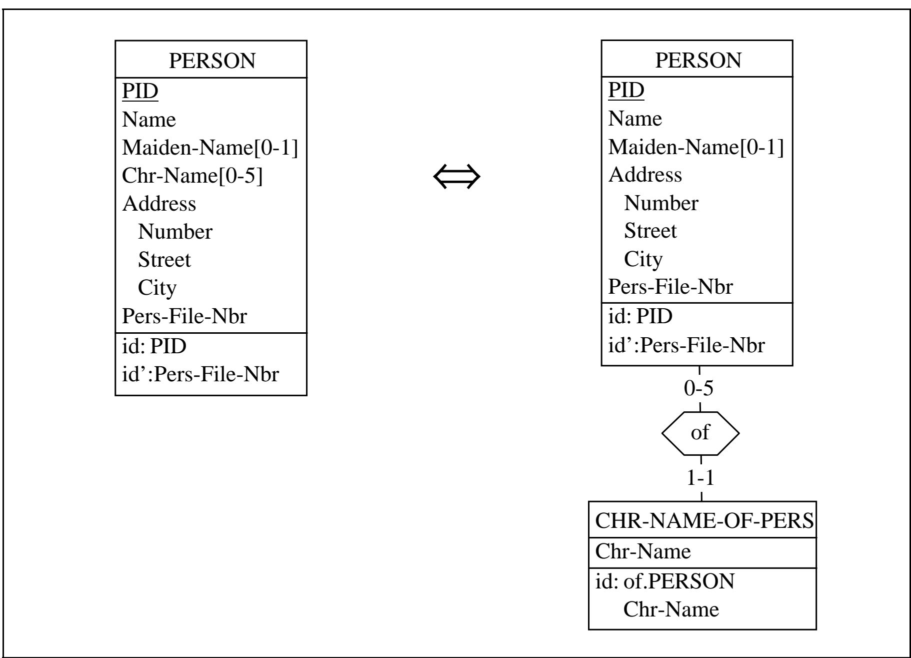


Figure 7.16 - Extracting attribute Chr-Name as an entity type through *Instance representation*.

Processing compound attributes

When the attribute to process is compound, the transformation also proceeds to its further decomposition, as it is the case for `ADDRESS`. In this example, we have chosen the *Value representation* technique (Figure 7.17). Can you guess what would have been the result with the *Instance representation* technique?

Processing identifier attributes

The next example concerns an attribute defined as an identifier of `PERSON` `(Pers-File-Nbr)`. Applying either technique (*Value* or *Instance representation*) gives the same result (why?) (Figure 7.18). Observe that, in both schemas, `Pers-File-Nbr` can be used to uniquely designate a `PERSON` entity, either directly (left) or indirectly (right).

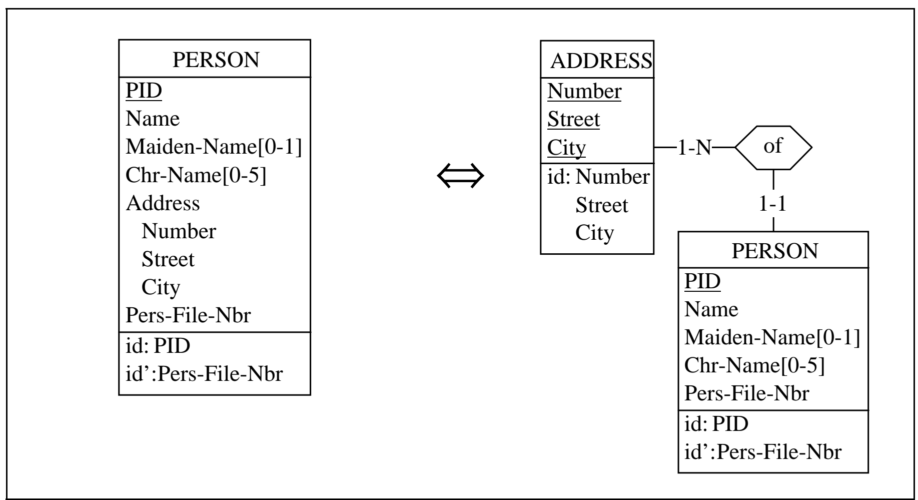


Figure 7.17 - Extracting attribute Address as an entity type through *Value representation*.

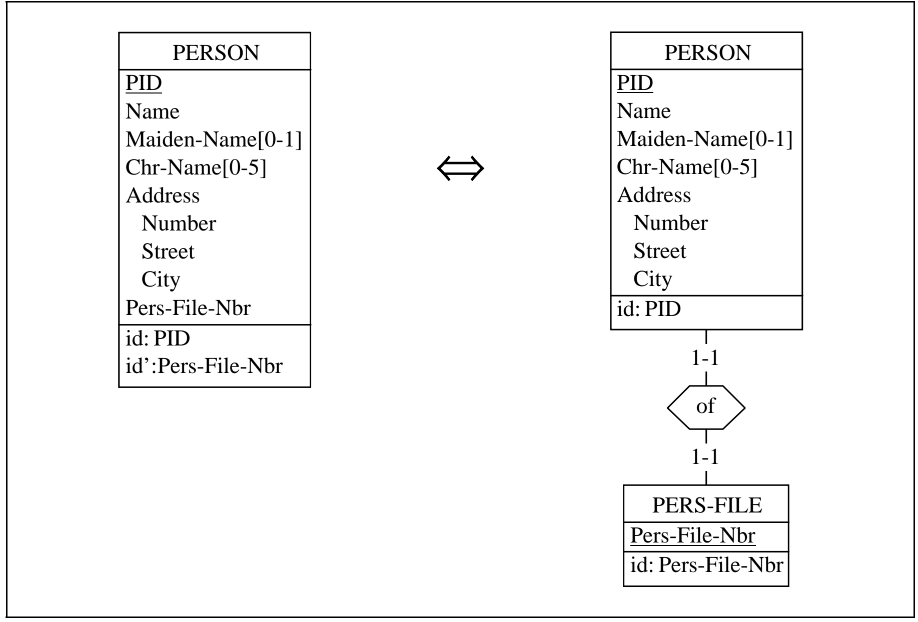


Figure 7.18 - Extracting attribute Pers-File-Nbr as an entity type through *Value* or *Instance representation*.

Processing components of an identifier

Let us now examine how the transformation behaves when applied on a component of an identifier (or of a group in general). In the example of Figure 7.19 (left), the identifier is made up of three components. Two of them are successively transformed into the entity types SUPPLIER and PRODUCT. The identifier has been modified accordingly.

These three schemas conveys exactly the same semantics, though through different presentations.

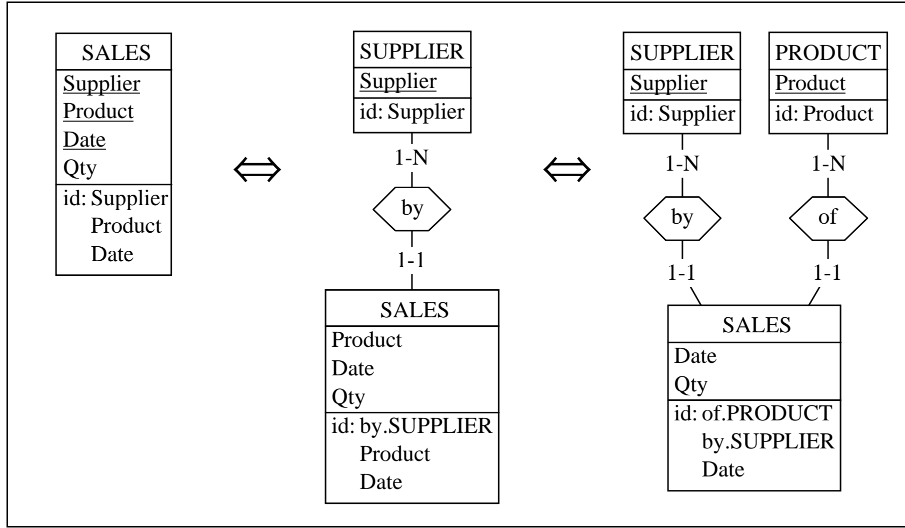


Figure 7.19 - Evolution of an identifier when its components are transformed into entity types.

Combined attribute transformations

The last example shows that combining several transformations can lead to new techniques. Let us consider once again the coexistence constraint discussed in Lesson 6. To get a better feeling of what this constraint exactly means, we gave an equivalent structure: an optional compound attribute, namely Marriage. Now, let us apply the *Attribute/Entity type* transformation to this compound attribute. We get a third schema (in fact two schemas, according to the technique chosen), which is strictly equivalent to the first one.

Therefore, we can consider a new transformation[2] which replaces a coexistence group by an entity type. It can be illustrated as follows:

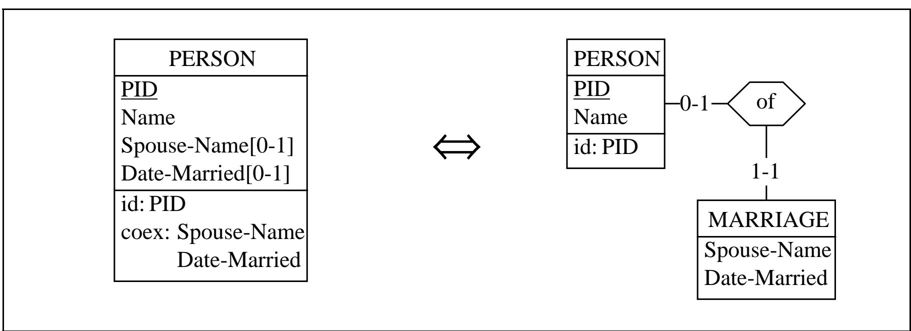


Figure 7.20 - Transformation of a coexistence constraint

2. In this composed transformation, we have chosen the *Instance representation* technique. Can you justify this choice?

Summary of Lesson 7

In this lesson, we have studied no new notions.

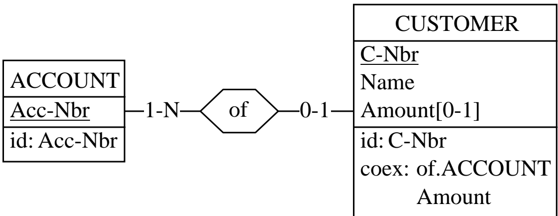
However, we have learned how

- to build a conceptual schema incrementally
- to transform an atomic attribute into a compound attribute by giving it a first component **New / Attribute / First att.**
- to transform an attribute into an entity type
  Transform / Attribute / -> Entity type
- to transform an entity type into an attribute
  Transform / Entity type / -> Attribute

Exercises for Lesson 7

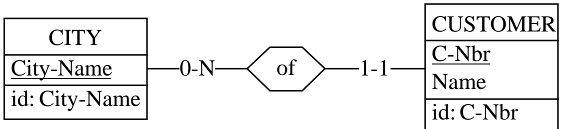
7.1 Consider the following schema. Propose another **equivalent** schema in which the coexistence constraint does not appear.

Note: the eraser cannot be considered a semantics-preserving transformation!

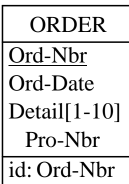


7.2 You find the following schema a bit too complex. Therefore, you propose to integrate `CITY` into `CUSTOMER` as an attribute. Try to do this through the *En-*

tity type/Attribute transformation. Can you explain the behaviour of DB-MAIN?



7.3 Suppose that, so far, the analysis has led to the following schema.

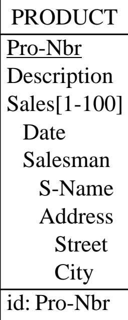


Add the constructs which represent these facts:

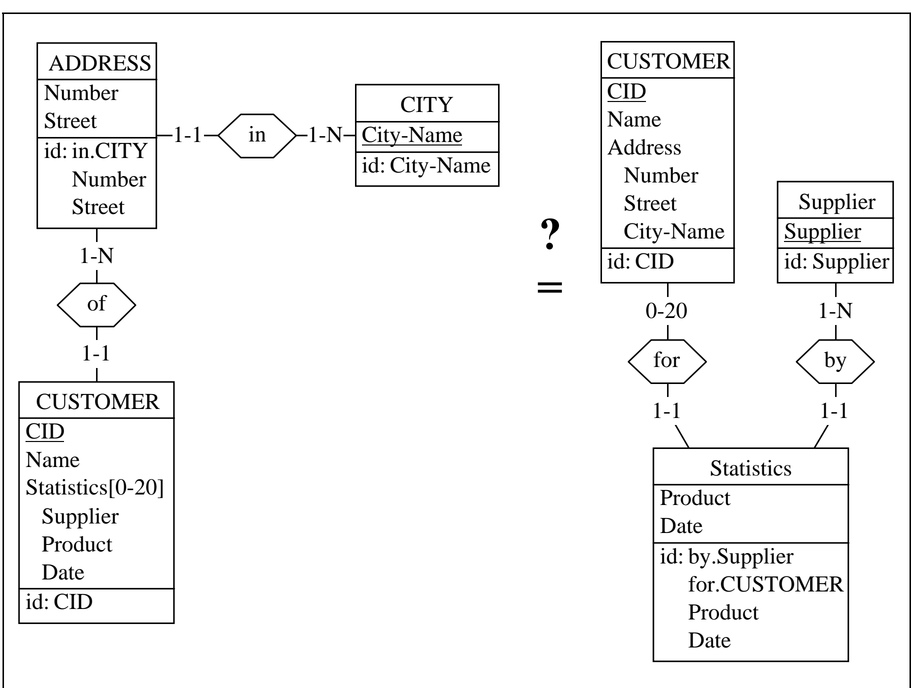
"*For each detail of an order, we know the quantity ordered. The order is characterized by the id of its customer. Each product has a unit price, a name, and a quantity on hand. Each customer has an id, a name, and an address. In a given order, there are no two details specifying the same product*".

7.4 A first analysis has produced the following schema. We want to build simple schemas only, i.e., schemas without compound attributes. Suggest an equivalent schema which satisfies this constraint. First, use *Attribute/Entity type* transformations only.

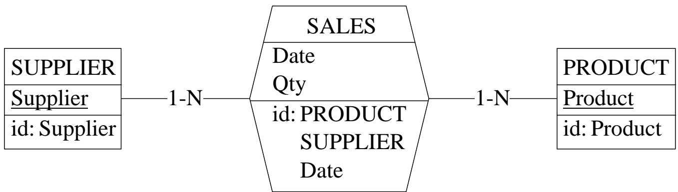
Then, try another solution by allowing also the disaggregation transformation (**Transform** / **Attribute** / **Disaggregation**). Which do you prefer? Why?



7.5 Your colleague thinks that these two schemas are equivalent. Do you agree with her?



7.6 Consider the following schema. One claims that it is equivalent to those of Figure 7.19. True or false? You could be wise to explore the transformation toolkit of DB-MAIN.



7.7 Let us go back to entity type `COPY` (Figure 7.11). We assume that attribute `State` is optional, and that `State-Comment` only exists if `State` has a value. In other words, the valid patterns are the following:

- `State` is void and `State-Comment` is void,
- `State` is not void and `State-Comment` is void,
- `State` is not void and `State-Comment` is not void.

Modify the structure of entity type `COPY` to represent this property explicitly.

Lesson 8

Conceptual Analysis (2)

Objective

Through this lesson, we will complete the analysis and design of the conceptual schema we initiated in Lesson 7. We will also go on discussing the important *Attribute/Entity type* schema transformation.

8.1 Starting Lesson 8

We start DB-MAIN and we open the project `concept-7.lun`. We immediately change the name of the project to `Concept-8` and save it as `concept-8`

8.2 The analysis

Let us remind the procedure followed so far: the text is decomposed into elementary sentences, each of them stating an elementary fact about the application domain. If this fact seems to be new, it is introduced in the current conceptual schema as new constructs, or as a modification of some parts of the schema.

8.3 The borrowers

(28) A copy can be borrowed, from a given date, by a borrower.

Analysis: so far, facts about the borrowings are represented by the optional attribute `BORROWER` of `COPY`. This sentence tell us more about the fact that a copy was borrowed by a borrower: this fact occurred on a given date. This information is about the action of borrowing itself, so we should make this action explicit, for instance as a relationship type

Being said in passing, have you observed that nouns often are represented by entity types - or attributes - and verbs by relationship types?

Action: Since we know that *borrowers borrow copies*, we first represent borrowers by an entity type (`BORROWER`), by transforming the attribute `Borrower` into an entity type. We select this attribute, then we execute command **Transform** / **Attribute** / **-> Entity type**. We give the entity type the name `BORROWER`, and the attribute the name, say, `Borrower-ID`. The rel-type will be named `borrowed`.

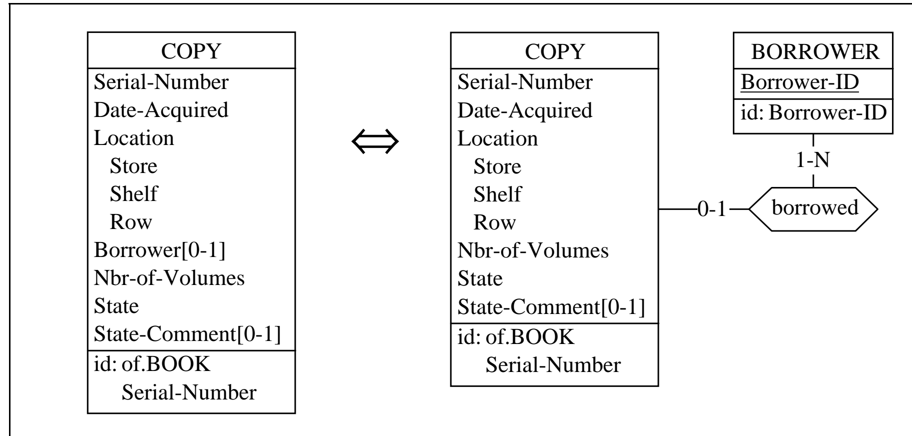


Figure 8.1 - Borrowers are explicitly represented by entity type BORROWER

Let us observe the cardinality `[1-N]` of `BORROWER`. It results from the fact that the left-side schema does not allow representing borrowers independently of their copies. Consequently, the right-hand schema does not represent borrowers without copies `[1-N]` either. The transformation normally ensures the strict equivalence of both left and right side schemas, and cannot introduce new constructs or modify the properties of the source schema. However, we could find more interesting to represent borrowers even when they currently borrow no books. Therefore, we change the cardinality into `[0-N]`.

Now, at last, we can represent the borrowing date by adding the attribute `Borrow-Date` to the rel-type `borrowed` (Figure 8.2)

(29) Borrowers are identified by a personal id.

Analysis: what we have called `Borrower-ID`, the identifying property of borrowers, is nothing else than their `Pid`.

Action: we change the name of attribute `Borrower-ID` into `Pid` (Figure 8.2).

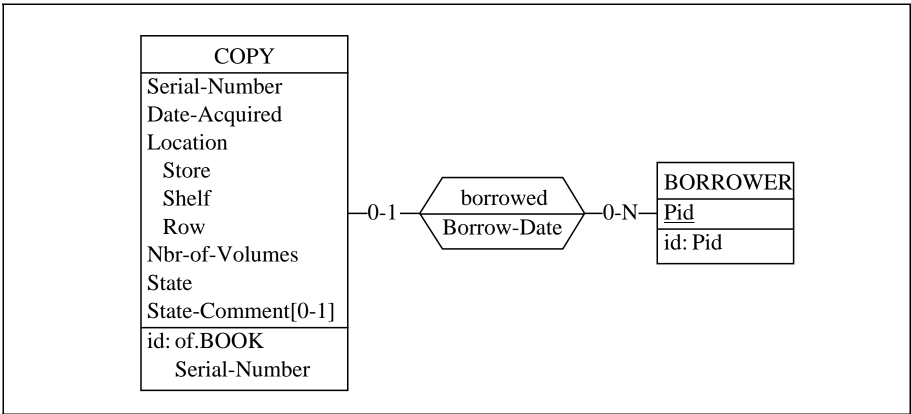


Figure 8.2 - Recording the borrowing date.

(30) The library records the name, [. . . of each borrower]

Analysis: each borrower has a name.

Action: we add attribute `Name` to `BORROWER`.

(31) the first name,

Analysis: they each have a first name as well.

Action: we add the attribute `First-Name`.

(32) the address

Analysis: ... and an address.

Action: we add the attribute `Address`.

(33) (name of the company, street, zip-code and city name),

Analysis: this address comprises the name of the company of the borrower, the street name, as well as the zip-code and the city name.

Action: we make `Address` a compound attribute as follows. We open the Attribute box of `Address`, and we click on the button First att., which opens the box of the first component of `Address`. We enter

the description of the attribute Company, then we click on Next att. to add the other components: Street, Zip-Code and City.

(34) as well as the phone numbers of each borrower.

Analysis: a borrower can have several phone numbers. How many? The text is vague on this, so that we could represent this fact by the attribute Phone[0-N]. On the other hand we could try to precise the information by asking it to the clercks of the library. A short phone call inform us that:

- ... a borrower MUST have a phone number, otherwise we cannot contact him when needed;
- and how many such numbers can they have?
- any number, but I don't remember any of them having more than three or four.
- would you agree on a maximum of five?
- sure!

So: PHONE[1-5]!

Action: we add such an attribute (Figure 8.3).

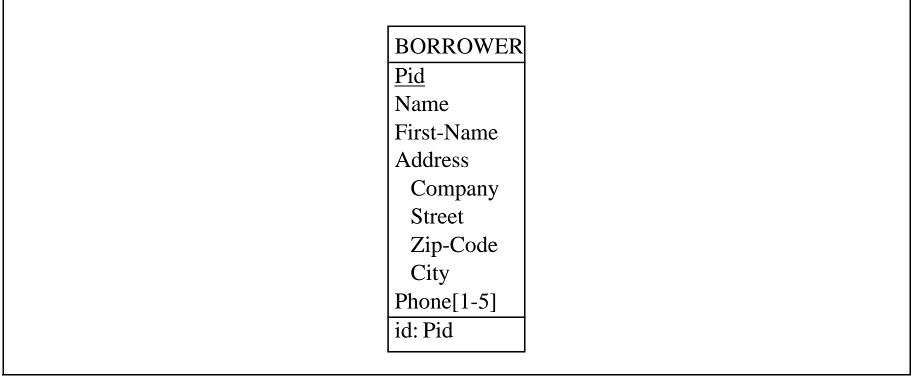


Figure 8.3 - Representation of phone numbers.

(35) In addition, when (s)he is absent, another borrower (who is responsible for the former) can be contacted instead.

Analysis: at first glance, this can be reworded as: each borrower can be associated with another borrower (his responsible). Two questions:

1. can someone be responsible for more than one borrower?
2. has each borrower a responsible?

We suggest that the answers are YES and NO respectively, but the second question deserves being discussed a bit further. If each borrower must have a responsible, then who will be the responsible of the first borrower? Perhaps himself? Obviously not. According to the intended purpose, a responsible must be a different person. On the other hand, the responsible for a given borrower may be unknown at the present time. So we must state that each borrower *can have* a responsible.

Action: we declare a cyclic, *one-to-many* relationship type named `responsible-for`. It has a `[0-1]` role named `borrower` (the borrower) and a `[0-N]` role named `responsible` (the responsible). Since the first role has the same name as the entity type, we decide not to name it explicitly (Figure 8.4).

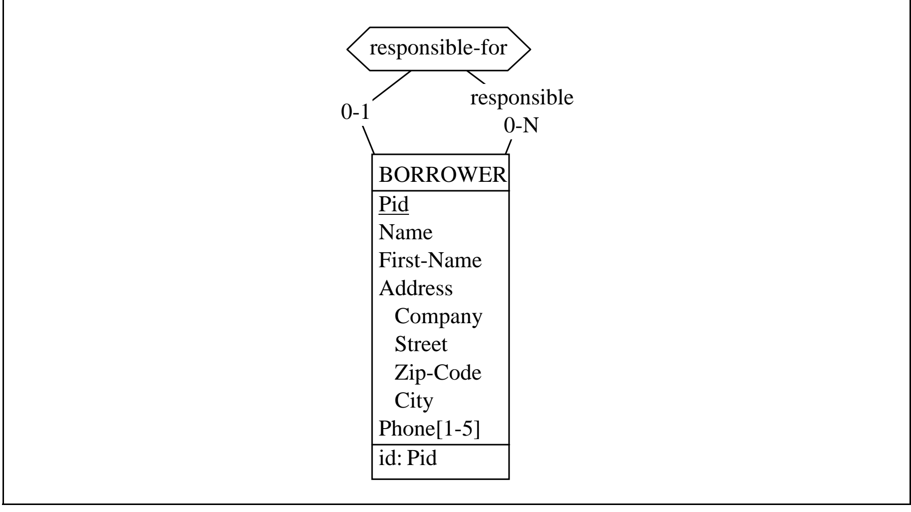


Figure 8.4 - Each borrower can be associated with another borrower, called his responsible.

8.4 Borrowings and projects

(36) A copy is borrowed on behalf of a project

Analysis: the information on the fact that a copy is borrowed is augmented by the project for which this copy has been borrowed.

Action: we add the attribute `Project` to the relationship type `borrowed` (Figure 8.5).

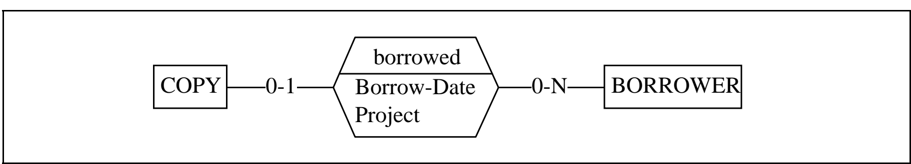


Figure 8.5 - Introducing Projects as an attribute of rel-type borrowed.

(37) (identified by its title, ...

Analysis: each project has a name, and all project names are distincts. This information does not contradict the current schema: the values of attribute `Project` are project titles.

Action: none

(38) ... but also by its internal code).

Analysis: now projects have several properties: they have names, they have codes and they are involved in borrowed copies. This is a bit too much to keep them as mere attributes. It would be better to make them entities, characterized by their names and their codes. In addition, names and codes identify the projects.

Action: the first thing to do is to transform attribute `Project` into an attribute: we select `Project`, then execute the command **Transform** / **Attribute** / **-> Entity type**. The attribute `Project` of entity type `PROJECT` is renamed as `Title` (Figure 8.6) and we add a second attribute `Pcode`. We define two identifiers, one made of `Title` and the other made of `Pcode`. We suppose that the code is the preferred identifier, so we make it the primary id of `PROJECT`.

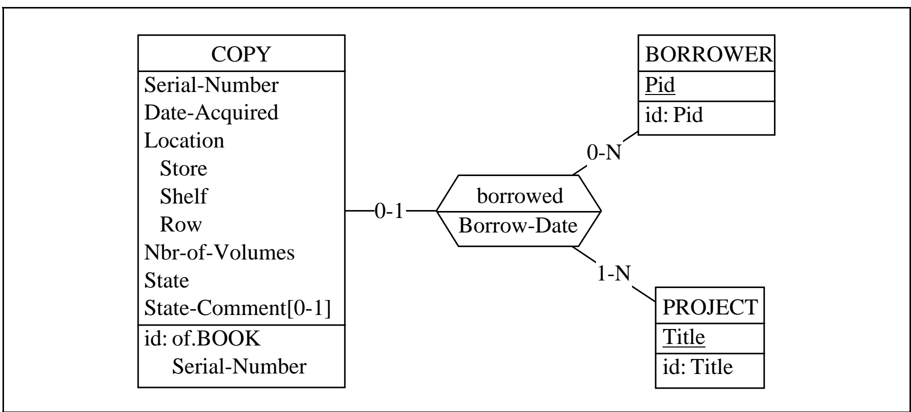


Figure 8.6 - Projects are represented through entity type PROJECT.

The transformation proposes the cardinality `[1-N]` for the role `PROJECT` (this fact will be discussed in the Addendum of this lesson). However we could find it better to change it into `[0-N]`, allowing projects to live without borrowings (Figure 8.7).

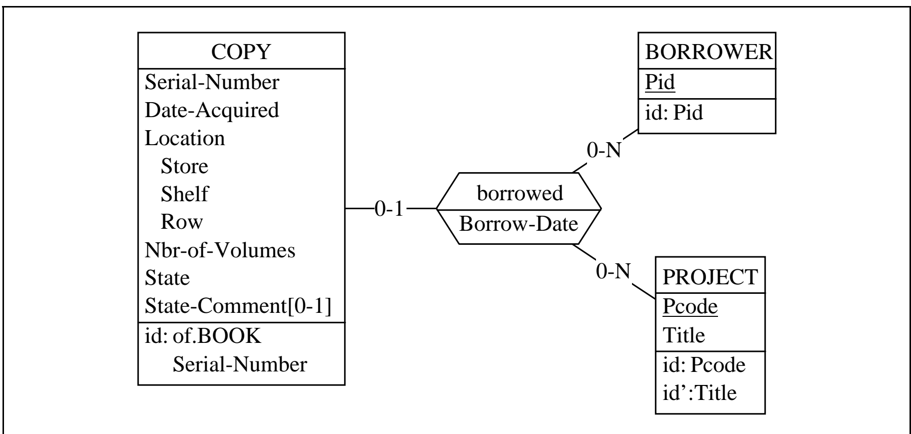


Figure 8.7 - Specifying the identifiers of PROJECT.

8.5 Borrowing history

(39) When a copy is brought back to the desk, the employee records the following information on this copy: the borrowing date, the current date, the borrower and the project;

Analysis: this sentence suggests that when a copy is brought back, some information is recorded about the borrowing. The copy, the borrower, the project and the borrowing date are known already. They are represented through the rel-type `borrowed`. The new information is the closing date,which is the current date.

Therefore, we could add the attribute `End-Date` to `borrowed`. This attribute is optional, since it has a value for closed borrowings and has no value for current borrowings. However, this is not sufficient. The cardinality `[0-1]` of `COPY` states that a copy can be borrowed only once at a given instant. If `BORROWED` now represents both current and closed borrowings, then we must admit that a given copy can have been borrowed more than once, in the past and currently, so that we must generalize the cardinality of `COPY` to `[0-N]`.

Unfortunately, this question of cardinality still is a bit more complex. Indeed, we should add the following constraint: if the attribute `END-DATE` of a relationship `BORROWED` has no value, then this relationship represents the fact that the copy is currently borrowed; this copy cannot appear in another relationship with no `End-Date` value. This constraint states that in the subset of the `borrowed` relationships with no `End-Date` values, the cardinality of `COPY` is reduced to `[0-1]`. Not really that simple! More on this later on.

Action: we add the optional attribute `End-Date` to `borrowing` (Figure 8.8) ... and we keep the complex constraint on the cardinality of `COPY` in our head (we could better write it in the SEM description of the role)!

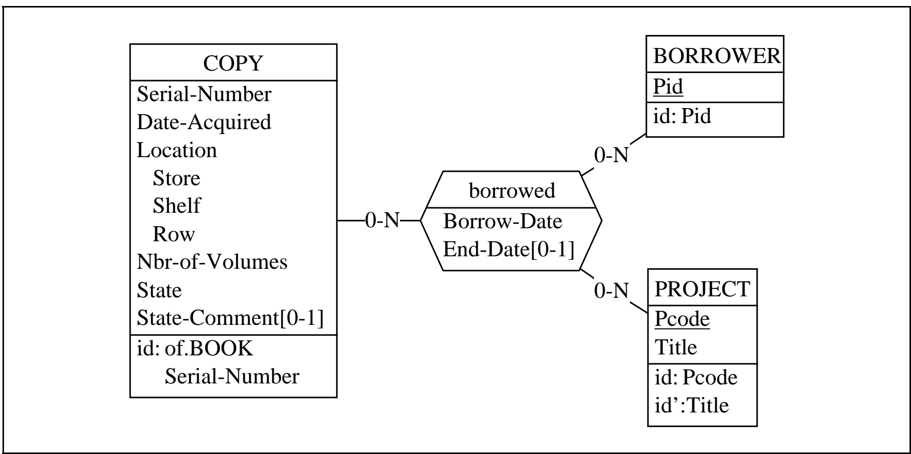


Figure 8.8 - A copy can have one current borrowing (with no End-Date value) but sveral closed boirrowings (with End-Date values).

(40) . . . then, the copy is put in a basket from which it is extracted at the end of the day to be stored in its location, so that it can be available again from the following day on.

Analysis: in short, a copy cannot be borrowed the day it was brought back. This property is not as simple as those we encountered so far. It tells us about time constraints related to sequences of events. Normally, such properties would require a richer model, allowing the representation of events, and of time constraints. However, we could try to express it as follows:

a copy must be borrowed on a date which is later than the latest date it has been brought back.

A possible translation could be: the `Borrow-Date` of the `borrowed` relationship which concerns the `COPY` entity `C`, and which has no `End-Date` value (i.e., the current borrowing of `C`), must be greater than the greatest value of `End-Date` of all the `borrowed` relationships in which `C` appears (i.e., the closed borrowings of `C`).

From this constraint, several properties can be inferred. For instance: all the `borrowed` relationships of a given `COPY` entity have distinct `Borrow-Date` (or `End-Date`) values. This property is weaker

than its origin, but is easier to express. Indeed, it can be expressed as an identifier of `borrowed` comprising `COPY` and `Borrow-Date`.

Due to the limited scope of this volume, we will prefer the latter formulation, despite the fact that it translates the true constraint only partially, and should be considered a poor substitute for it.

Action: we declare this identifier of `borrowed`.

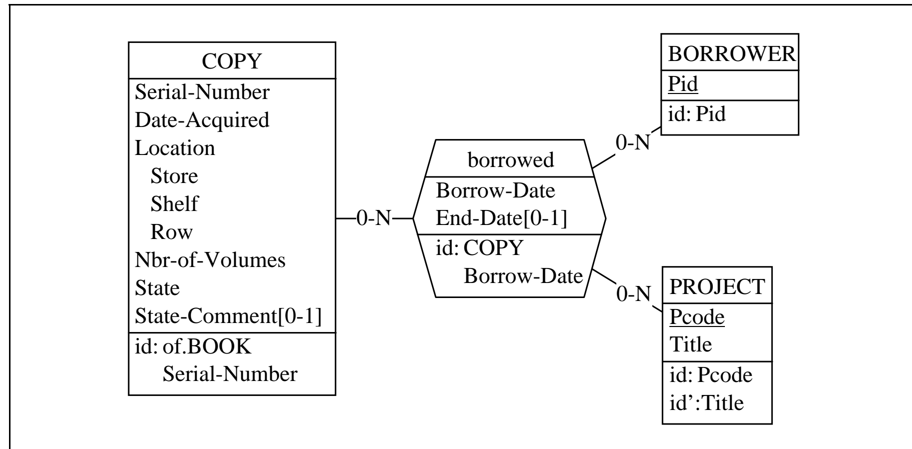


Figure 8.9 - Attempting to express the fact that copies can be borrowed only the day after they were brought back.

Discussion

This solution is worth being analysed a bit further. Indeed, the `borrowed` relationship type is submitted to several complex constraints. Most of this complexity comes from the fact that this rel-type represents two kinds of borrowings, namely the current borrowings and the closed borrowings. In addition, these two classes of concepts have different behaviours and usage profiles:

- current borrowings have a short life, they appear then disappear in a few weeks, while closed borrowings live much longer;
- current borrowings are created and deleted, while closed borrowing are created but never deleted;
- most current borrowings are consulted several times, while the closed borrowings are used very unfrequently (if any);

- current borrowings are submitted to management functions, while the closed borrowings are only used for statistical analysis;
- after a while, there are much more closed borrowings than current borrowings.

All this suggests representing these two classes by two distinct relationship types, one that represents the current borrowings, and the other representing the closed borrowings. So, we go back to the former representation of current borrowings (Figure 8.7), and we define a new relationship type, `closed-borrowing`. To normalize the names, we give `borrowed` the new name `borrowing`.

In this splitting, the complex identifier is associated with `closed-borrowing` only. This weakens this constraint still more, but the result is fairly easy to express (Figure 8.10).

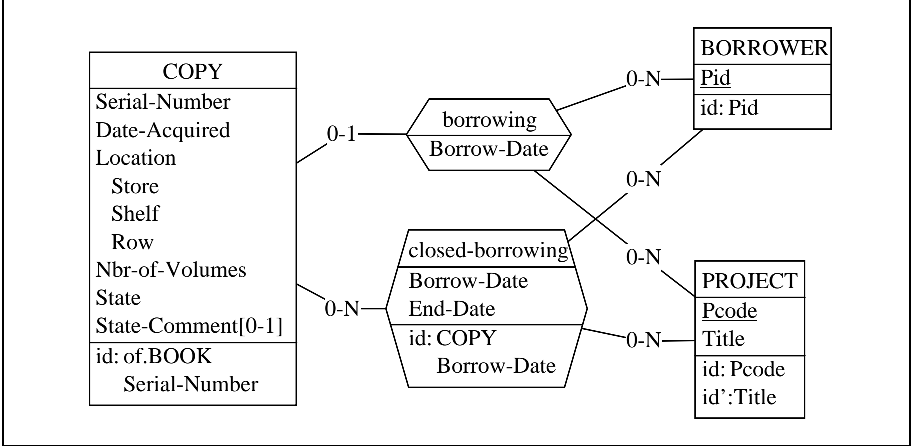


Figure 8.10 - Distinguishing current and closed borrowings.

8.6 The final schema

The schema can be considered as completed. It appears as in Figure 8.11.

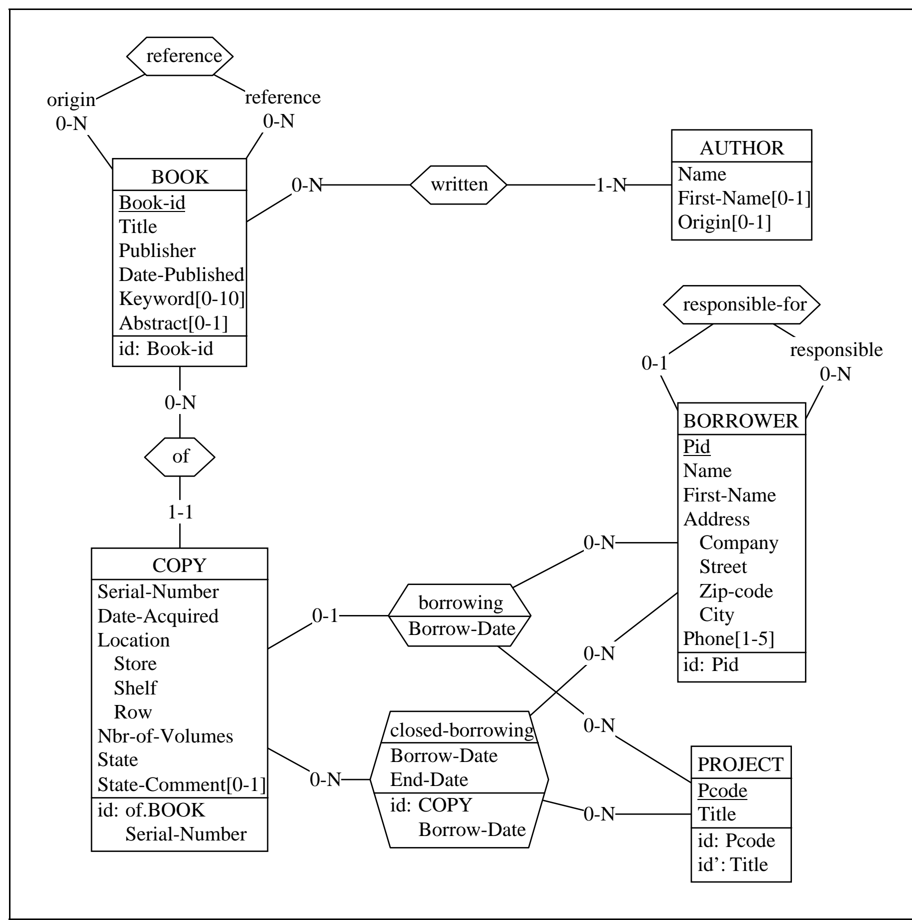


Figure 8.11 - The final schema.

We must verify that all the attributes have correct types and lengths. In addition, we should add to each object a description which can be represented by a SEMantic description (button SEM in each property box). This material could be made of excerpts from the text used to build the schema.

The textual extended version of the schema could appear in Figure 8.12.

```
                Schema LIBRARY/Conceptual / LIB [ST]

AUTHOR / AUT [S]
  Name char (30)
  First-Name[0-1] char (30)
  Origin[0-1] char (30) [S]
BOOK / BOOK [S]
  Book-ID numeric (6)
  Title char (30)
  Publisher char (40)
  Date-Published date (6)
  Key-Word[0-10] char (30) [S]
  Abstract[0-1] char (80) [S]
  id: Book-ID
BORROWER / BER [S]
  Pid char (6)
  Name char (30)
  First-Name char (30)
  Address compound (124) [S]
       Company char (40) [S]
       Street char (40)
       Zip-Code numeric (4)
       City char (40)
  Phone[1-5] numeric (10) [S]
   id: PID
COPY / COPY [S]
  Serial-Number numeric (6) [S]
  Date-Acquired date (10)
  Location compound (6) [S]
       Store numeric (2)
       Shelf numeric (2)
       Row numeric (2)
  Nbr-of-Volumes numeric (3) [S]
  State char (10) [S]
  State-Comment[0-1] char (80) [S]
  id: of.BOOK, Ser-Number
PROJECT / PRO [S].
  Pcode char (6)
  Title char (30)
  id: Pcode
  id’: Title
```

```
reference / REF [S] (
  origin [0-N]: BOOK [S]
  reference [0-N]: BOOK [S] )
borrowing / BING [S] (
  [0-1]: COPY
  [0-N]: BORROWER
  [0-N]: PROJECT
  Borrow-Date date (6) )
closed-borrowing / CLO [S] (
  [0-N]: COPY
  [0-N]: BORROWER
  [0-N]: PROJECT
  Borrow-Date date (6) [S]
  End-Date date (6) [S] )
  id: BORROW-DATE, COPY
of / OF [S] (
  [0-N]: BOOK
  [1-1]: COPY )
responsible / RESP [S] (
  [0-1]: BORROWER [S]
  responsible [0-N]: BORROWER [S] )
written / WRIT [S] (
  [1-N]: AUTHOR
  [0-N]: BOOK )
..
```

Figure 8.12 - The text expression of the final schema. The role clauses of entity types have been removed for conciseness reasons.

8.7 Quitting the lesson

We save the current project under the name `concept-8`), and we quit DB-MAIN.

Technical addendum

8.8 Discussion on the *attribute/entity type* transformation (continued)

In this lesson, we have used an extended version of the transformation discussed in lesson 7 that processes rel-type attributes as well. This version is worth being described by some representative applications.

Let us start with the following schema, expressing that products are manufactured for markets by companies according to specified ratios. For simplicity, each entity type has an identifying attribute which has been given the name of the entity type. In actual situations, these attributes would have other names.

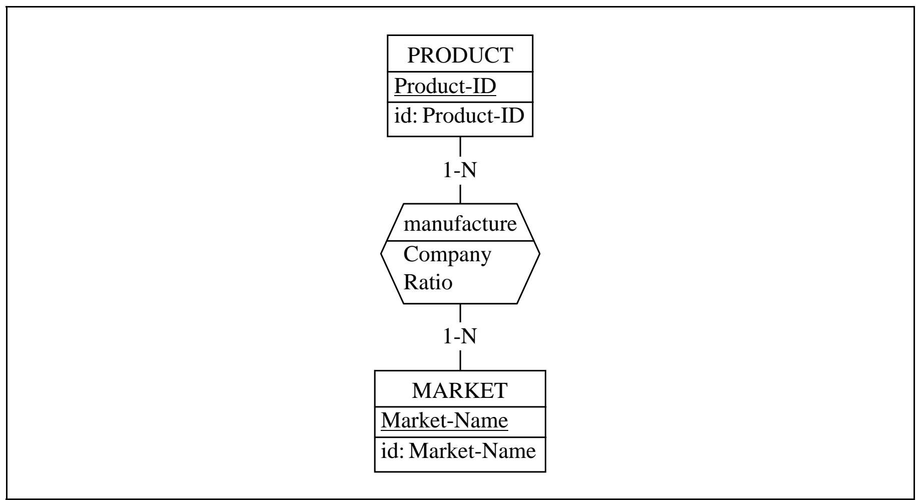


Figure 8.13 - A binary relationship type with attributes.

In the same way as we did with `PROJECT` in this lesson, we can extract the `COMPANY` attribute to replace it by an equivalent entity type. We select this attribute and we ask for its transformation into an entity type as usual.

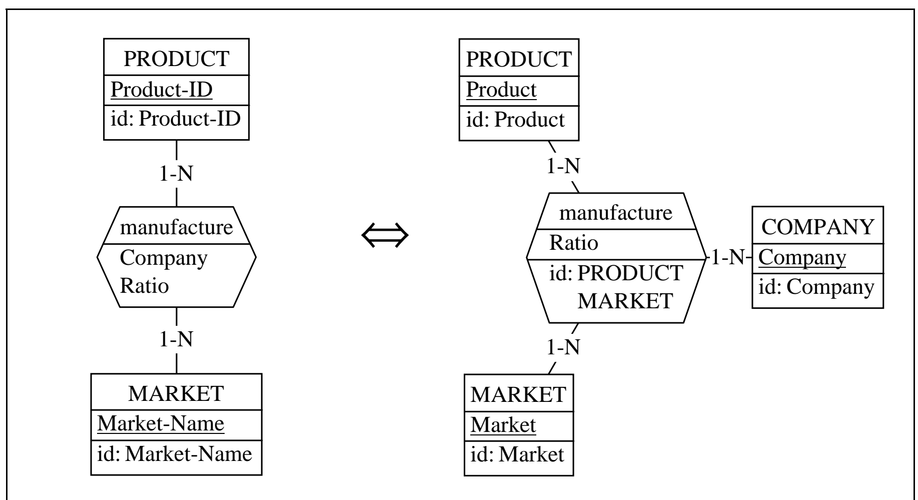


Figure 8.14 - Extracting an attribute to form as new role.

The resulting schema exhibit some interesting characteristics.

First, quite naturally, the attribute `COMPANY` has disappeared, and has been replaced by the entity type `COMPANY`.

Secondly, `manufacture` has be given a third role, and is now of degree 3.

Thirdly, `manufacture` has got an explicit identifier. Where does it come from? In the left-side schema, `MANUFACTURE` has no declared identifiers. This means that it has a default (i.e., undeclared) identifier made of all its roles, namely `PRODUCT` and `MARKET` (see Lesson 4). The attribute `Company` is no part of this identifier, otherwise it should have been declared explicitly. As a result of the transformation, the composition of the rel-type has changed, and the identifier can no longer be considered implicit (the implicit id would comprise all the roles). Hence the declared identifier, making both schemas fully equivalent.

Finally, we observe that the cardinality of the new role is `[1-N]`, and not `[0-N]` as it would be thought at first glance. DB-MAIN is right, and our first feeling was wrong. Indeed, a `[0-N]` cardinality would have meant that some companies could exist without being involved in manufacturing products for markets. Such a situation would be quite natural, but it cannot be represented

in the left-side schema, in which companies can only be represented when they appear in manufacture relationships.

Now, let us practice the reverse transformation by reducing, say, entity type MARKET to a mere attribute. We select the entity type MARKET, and we execute the command **Transform** / **Entity type** / **-> Attribute**. The evolution of the identifier is worth being examined.

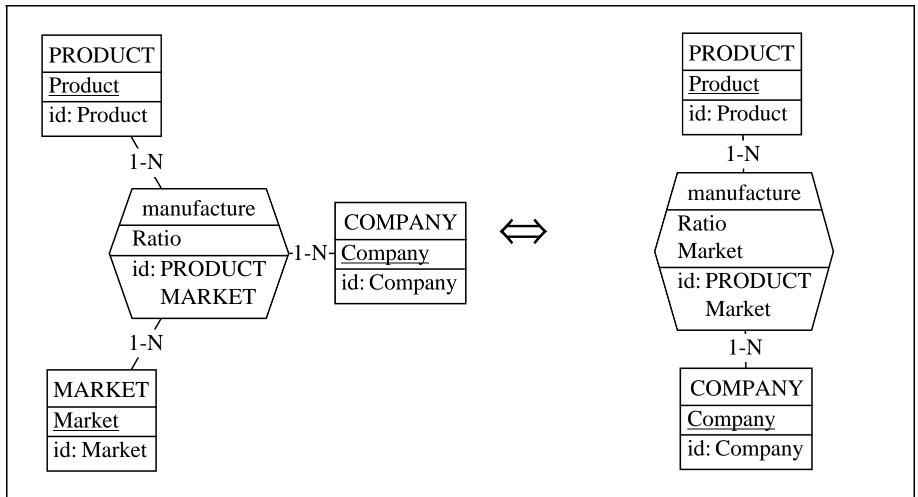


Figure 8.15 - Reducing a role to an attribute.

Other equivalent versions of the source schema can be derived by the application of these transformations.

Summary of Lesson 8

In this lesson, we have encountered some interesting situations:

- complex time constraints, and how to get rid of them (thus degrading the quality of the schema by tolerating weaker constraints);
- when to replace `[1-N]` cardinalities, resulting from reversible transformations, by more general cardinalities `[0-N]`;
- an entity type with two identifiers (`PROJECT`);
- a relationship type with an explicit identifier.

We have also gone in further detail about how

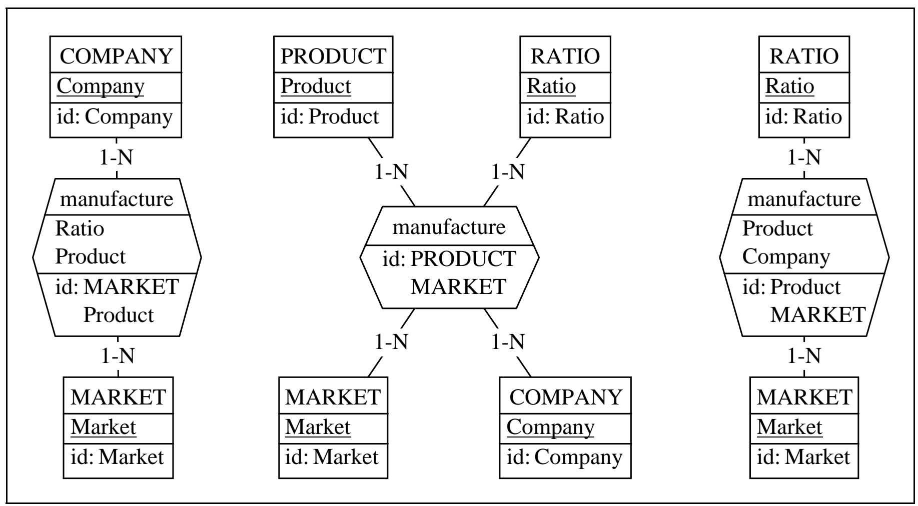
- to transform an attribute of a rel-type into an entity type, therefore increasing the degree (number of roles) of the rel-type:

  Transform / Attribute / -> Entity type
- to transform a role of a rel-type into an attribute, therefore decreasing the degree (number of roles) of the rel-type:

  Transform / Entity-type / -> Attribute

Exercises for Lesson 8

8.1 Do you think that each of the following three schemas can be obtained by transforming the schema of Figure 8.13?



8.2 **ORM / Entity-relationship conversion**

The Entity-relationship (ER) model, together with all its variants, is probably the most widespread formalism to specify conceptual database schemas. However, several other models exist with the same purpose. One of the most interesting of them is the Object-Role model, a variant of the NIAM model, particularly suited to precise conceptual analysis[1]. In this model, there are two kinds of objects, namely the *lexical object types* (LOT) and the *non lexical object types* (NOLOT). A LOT represents a class of printable symbols (such as `NAME`, `COLOUR`, `LENGTH`) while a NOLOT represents a class of abstract objects (`CUSTOMER`, `PRODUCTS`, `ADDRESS`). A LOT and a NOLOT can be associated through a *naming bridge* and two NOLOTs can be associated through an *idea bridge*. Each bridge is a binary relationship type (possibly N-

1. see Halpin,T., *Conceptual Schema and Relational Database Design*, Prentice-Hall, 1995, ISBN 0-13-355702-2 . Consult also `http://www.inconcept.com`.

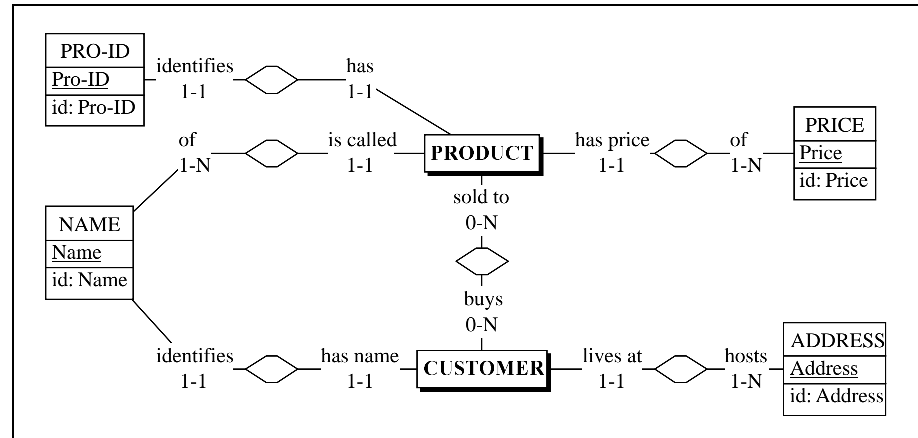
ary for ideas in some models), with which some constraints are associated: identifiers, totality, exclusiveness, etc. The bridges have no names, but each role has a name which suggests the semantic link from the role object type to the other object type(s).

The ORM graphical representation is quite specific, but it is possible to mimic an ORM schema with Entity-relationship constructs. An ORM schema can be represented as follows:

- LOT A is represented by entity type A, which has one identifying attribute named A too;
- NOLOT E is represented by entity type named E, without attributes;
- a naming bridge is represented by a binary rel-type between a LOT entity type and a NOLOT entity type;
- an idea is represented by a rel-type between two or more NOLOT entity types.

To make the schema more realistic, we have left the rel-types unnamed[2], and given each role a meaningful name.

For instance, the following schema is the Entity-relationship expression of an ORM schema[3]



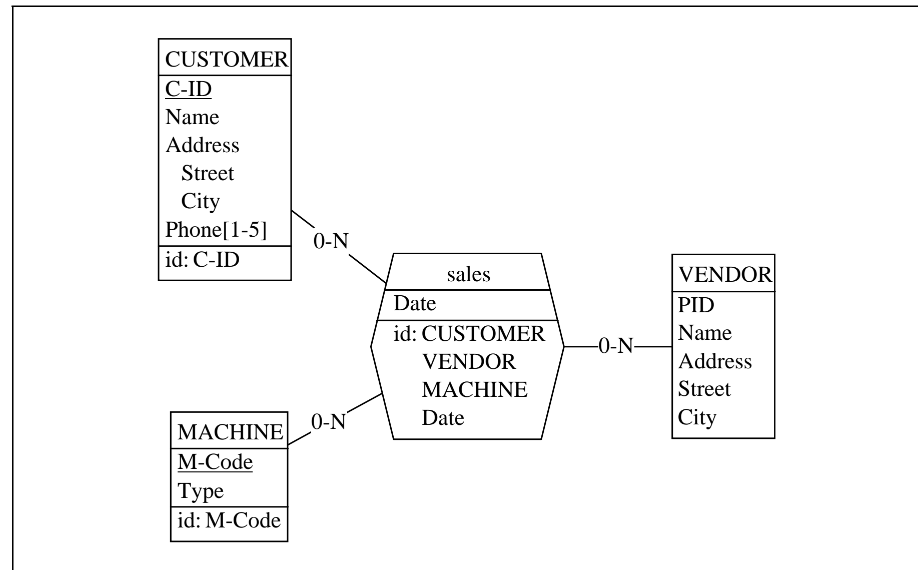
2. The DB-MAIN tool has a special feature for this: any object name can include the symbol "|", in which case this symbol as well as the following characters are not displayed.
3. To highlight some objects in a schema, such as PRODUCT and CUSTOMER, select them then click on the button Mark in the Standard tool bar.

The ER expression of an ORM schema offers the following characteristics:

- the rel-types have no attributes,
- some entity types (NOLOT) have no attributes,
- all the other entity types (LOT) have one attribute, which is its identifier, and which has the same name as its entity type.

Question: What is the source ER schema which the above ORM-like schema is an expression of? Rebuild the source ER schema through schema transformations.

8.3 Transform the following schema into an ORM-like schema. Use schema transformations (you could need some operators currently lacking in the tool: could you suggest some new schema transformations?).



Lesson 9

Logical Design

Objective

This (long) lesson discusses how a conceptual schema can be transformed into relational structures. Such a translation is called *Logical design*. It also defines the concept of SQL-compliant schema, i.e., a schema whose constructs can be explicitly and directly expressed into SQL structures. Some new powerful relationship type transformations are presented.

9.1 Starting Lesson 9

We start DB-MAIN and we open the project `concept-8` in which we have built the conceptual schema of our case study. We change the name of the project name into `Logical-9` and we save it as `logical-9`.

9.2 Logical design

All along lessons 7 and 8, we have carefully built the conceptual schema of a small library. This schema is a real piece of art which deserves being admired much longer than we did so far. However, it is an abstract piece of art. Being said more practically, it does not work! Feeding a DBMS[1] with the `*.lun` expression of this schema is useless. DBMS only understand SQL texts (or any other similar material), and are therefore not overly impressed by our performance in the previous lessons.

Our objective now is a little clearer: it is to translate this conceptual schema into the equivalent SQL text. Let us examine in some detail the idea before developing this translation process.

Asserting that our schema is nice is not (only) an aesthetic claim. It means that the schema expresses correctly, and in an elegant way, all the meaning included in the starting text (and more generally in any information source we could have used), and nothing else. Specialists say that this schema formalizes the *user's requirements* of the future database. This property is more generally called *correctness*: the conceptual schema is correct if it expresses all the user's requirements.

The concept of equivalent SQL text is an important one too. It implies that this text is an *operational expression*[2] of all the specifications included in the conceptual schema, and nothing more.

---

1. A DBMS is a *Database Management System*. ORACLE, SYBASE, DB2 and SQL-Server are relational DBMS which understand some dialect of the SQL language. Non-relational DBMS do exist as well. IMS, IDMS, DATACOM/DB, TOTAL or IMAGE are such DBMS, generally used as the data engine of legacy systems. Even plain programming languages offer data management systems, generally called *File Management Systems*. When we do not want to distinguish these categories, we talk about *Data Management Systems*, or DMS.
2. I.e., an expression which can be operated or processed by a software.

You could probably think that we are a bit fussy about this problem. Indeed, producing the SQL text of a conceptual schema is not that complicated: it suffices to execute the **Transform** / **Quick SQL** command (see Lesson 1). Right, but such a procedure wouldn't tell us anything on exactly how the resulting SQL structures have been obtained. Knowing this is important to understand a not-so-elementary translation process, in such a way that we can control it more effectively when needed, that is, when coping with more complex databases. The automatic **Quick SQL** procedure is basically an unsophisticated way to get a rather naive relational database. It is not a bad procedure, but should we consider additional requirements such as space or execution time minimization, the schema resulting from this oversimplistic translation would most probably be highly unsatisfactory, whatever the tuning effort you carry out afterwards. However, producing efficient database structures is a complex task that is clearly beyond the scope of this introductory volume.

In this lesson, and in Lessons 10 and 11, we will learn how to get a correct relational database structure which is equivalent to a conceptual schema. We will proceed in two steps. Through the first one, called **logical design**, we will obtain a logical schema, i.e., a schema representing tables, columns, primary and foreign keys, as well as other constraints. The second step, called **physical design**, will augment the logical schema with physical specifications such as index and files, therefore producing the physical schema, and will generate the corresponding SQL text.

This process can be sketched as in Figure 9.1.

In Lesson 9, 10 and 11, we will study how to perform the logical design of a database. We will propose a simple procedure only, similar to those carried out by the **Transform** / **Quick SQL** and **Transform** / **Relational model**, but that will give us an clear understanding of how the final result will be obtained. Lesson 12 will cope with physical design and SQL generation.

9.3 The concept of *Relational Logical Schema*

A Entity-relationship schema can be called *SQL-compliant* if each of its components can be *directly* and *explicitly* represented by a relational object in a one-to-one way. Such a schema will also be called a *relational logical schema*.

For instance, the entity type `COMPANY` can be represented by the table `COMPANY`, the attribute `Com-Address` can be represented by the column `Com-`

Address, the reference attribute Pro-ID can be represented by declaring Pro-ID a foreign key, etc.

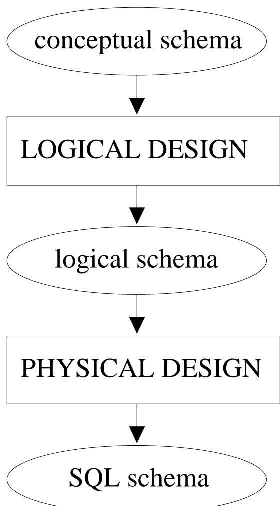


Figure 9.1 - The 2-step process that transforms a conceptual schema into a SQL schema.

On the contrary, a relationship type, a multivalued attribute, or a supertype/subtype structure have no direct representation in the relational model, and should not appear in any so-called relational logical schema. In short, a 8-year old child should be able to translate any relational logical schema in SQL-DDL without particular effort. For instance, the schema of Figure 9.2, borrowed from Lesson 3, is SQL-compliant.

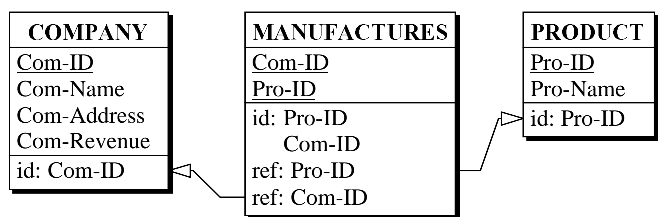


Figure 9.2 - A SQL-compliant schema made up of tables, columns, primary and foreign keys.

Indeed, it describes 3 tables, 8 columns, 3 primary keys and 2 foreign keys. Its SQL translation is immediate and does not require any particular skill:

```
create database Manufacturing;

create table COMPANY (
     Com-ID char(15) not null,
     Com-Name char(25) not null,
     Com-Address char(50) not null,
     Com-Revenue numeric(12) not null,
     primary key (Com-ID));

create table PRODUCT (
     Pro-ID char(8) not null,
     Pro-Name char(25) not null,
     primary key (Pro-ID));

create table MANUFACTURES (
     Com-ID char(15) not null,
     Pro-ID char(8) not null,
     primary key (Pro-ID,Com-ID),
     foreign key (Com-ID) references COMPANY
     foreign key (Pro-ID) references PRODUCT);
```

The concept of SQL-compliant schema

What is exactly a SQL-compliant schema? According to the definition proposed hereabove, it can be grossly defined as follows:

A **SQL-compliant** schema comprises only:
- entity types - single-valued and atomic attributes - identifiers - reference attributes

In other words:

A **SQL-compliant** schema **does not** comprise:
- IS-A relations - relationship types - compound attributes - multivalued attributes

The interpretation of a SQL-compliant schema into SQL concepts is immediate, and can be summarized by the following translation table:

DB-MAIN objects	SQL objects
entity type	table
attribute	column
primary identifier	primary key
secondary identifier	unique constraint
reference attribute(s)	foreign key

Of course, this table is not complete, but it is quite sufficient to tackle the logical design process.

9.4 Transformational approach to Logical design

Obviously enough, the hard job is not to translate SQL-compliant structures in SQL (remember, an 8-year old kid job), but to derive the logical schema from the conceptual schema, i.e., to transform the conceptual schema into an SQL-compliant schema.

For instance, the logical design aims at transforming in a systematic way the well known conceptual schema of Figure 9.3 into the SQL-compliant logical schema of Figure 9.4.

In this example, the translation is not particularly painful, but we can imagine that processing actual conceptual schemas, including a lot of compound, mul-

tivalued attributes, and of N-ary rel-types, could be somewhat more complex, and deserves more development.

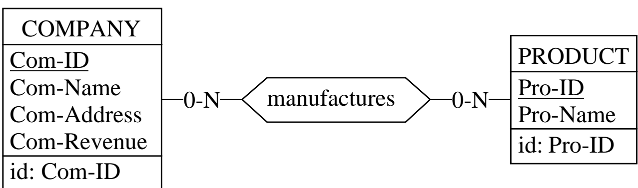


Figure 9.3 - A small conceptual schema ...

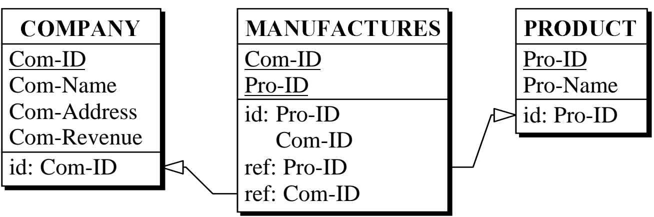


Figure 9.4 - . . . and its logical relational equivalent.

To help us in this translation process, we will use very powerful tools, namely **schema transformations**. We are already fairly acquainted with some of these operators. For instance, we know how to disaggregate compound attributes, or how to replace an attribute by an equivalent entity type, and conversely. However, this basic set is not sufficient. We need more techniques to carry out the logical design of a real database. Nevertheless we can already consider that the logical design can be considered as applying selected schema transformations on the source conceptual schema until it becomes fully SQL-compliant.

Two major questions arise:

1. what transformations do we need?
2. in what order must we apply these transformations to process any conceptual schema?

The first question will be addressed in Lessons 9 and 10, while the second question will be discussed in Lesson 11.

9.5 Dealing with *one-to-many* relationship types

Let us consider this elementary structure through the schema of Figure 9.5.

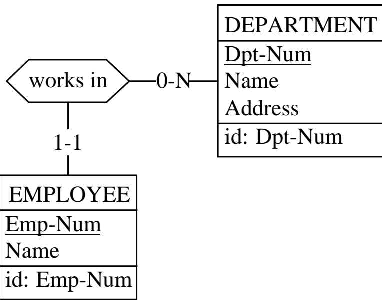


Figure 9.5 - A *one-to-many* relationship type.

The entity types and all the attributes are SQL-compliant and can be represented by tables and columns. There is only one invalid construct, namely the `works in` relationship type.

Representing a *one-to-many* rel-type is fairly easy: we add to `EMPLOYEE` the reference attribute `Dpt-Num` towards `DEPARTMENT`. In this way, one can retrieve the department an employee works in, and one can retrieve all the employees who work in a given department.

So we get the fully SQL-compliant logical schema of Figure 9.6.

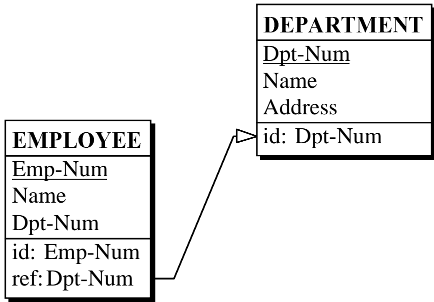


Figure 9.6 - Expressing a *one-to-many* relationship type through a foreign key.

In fact, the latter schema can be obtained automatically, through a specific schema transformation which replaces each *one-to-many* rel-type by reference attributes. We select the rel-type `works in`, and we execute the command **Transform / Rel-type / -> Attribute**.

So, we are provided with a tool which can process all the *one-to-many* rel-types of our schemas. Let us try it on another example. We consider the conceptual schema of Figure 9.7.

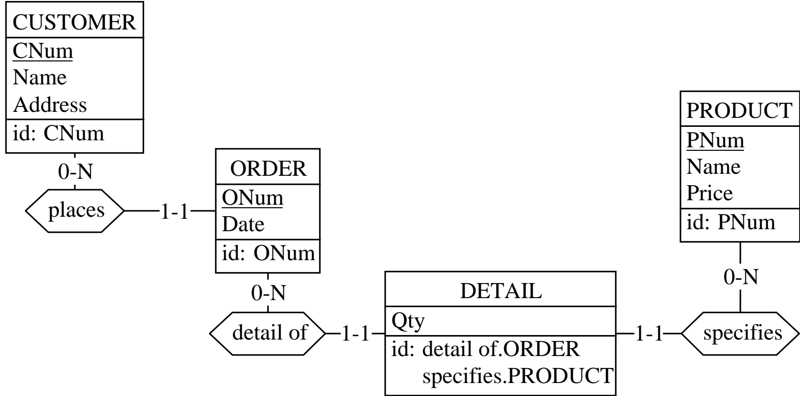


Figure 9.7 - A more comprehensive schema including *one-to-many* rel-types.

By transforming `places`, `detail of` and `specifies`, we get the logical schema of Figure 9.8.

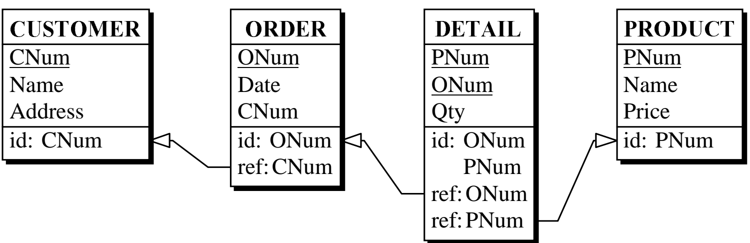


Figure 9.8 - . . . and its relational expression.

These transformations show how the role components of the identifier of entity type `DETAIL` have been replaced by reference attributes (or foreign keys) `ONum` and `PNum`.

This transformation is quite able to process *one-to-one* rel-types as well, as illustrated in the schema of Figure 9.9.

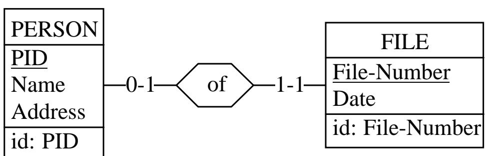


Figure 9.9 - A *one-to-one* relationship type.

... which is thoroughly transformed as in Figure 9.10.

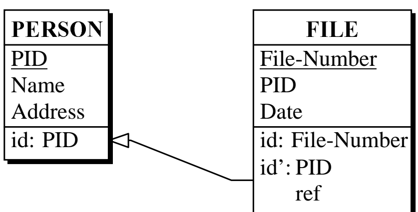


Figure 9.10 - Expressing a *one-to-one* relationship type through an identifying foreign key.

We can make three observations:

1. the foreign key `PID` of `FILE` is an identifier as well; indeed, there can be only one file per person, and therefore only one file per `PID` value. This identifier have been considered secondary since there already is a primary id.
2. the foreign key has been added to `FILE`, and not to `PERSON`, otherwise, the foreign key would have been optional, a situation designers do not like too much; DB-MAIN knows this, and has chosen to add the foreign key to the [1-1] side. But what about *bi-optional* rel-types, both roles of which are optional? Let us just try it (Figure 9.11).

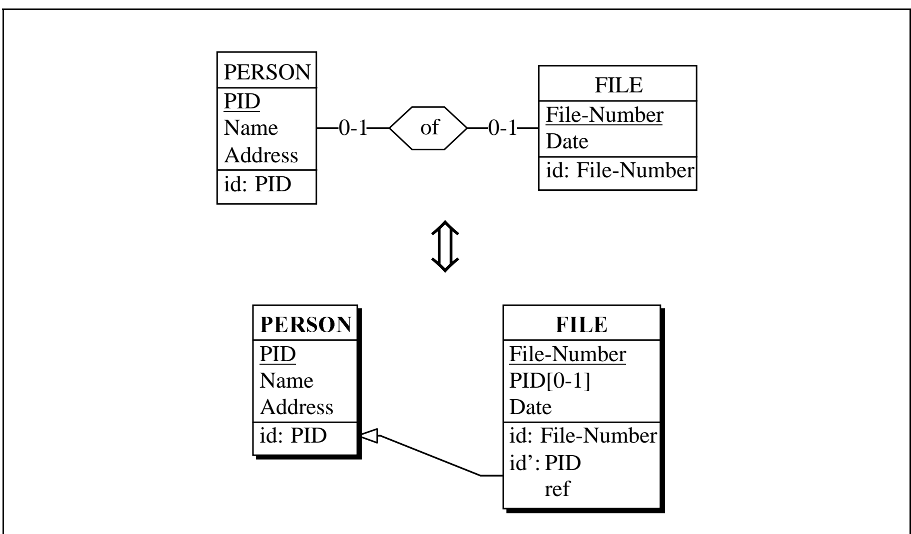


Figure 9.11 - Expressing an *bi-optional* rel-type.

Since there are two [0-1] roles, DB-MAIN is forced to make the foreign key optional, whether you choose to assign it to the table `PERSON` or to the table `FILE`.

Just like all the transformations studied so far, this one has an inverse (see Lesson 6). You can experiment this fact easily by recovering the origin schema:

1. select the foreign key by clicking on its group (not on the attribute it comprises!),
2. execute **Transform** / **Group** / **-> Rel-type**, call the new rel-type `of`.

9.6 Processing *many-to-many* relationship types

You probably think we have a straighforward solution to get rid of such rel-types: translating them into a table with two foreign keys, just in the way `manufactures` was processed at the beginning of this lesson. Yes and no!

Yes, this could be a nice solution, but we will not adopt it. We will prefer an indirect, but more general procedure through which the rel-type will first be transformed into an entity type + 2 *one-to-many* rel-types.

Let us apply such a transformation on `manufactures` (Figure 9.12).

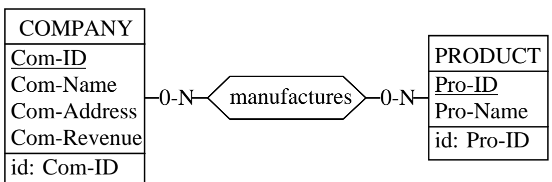


Figure 9.12 - A *many-to-many* relationship type.

We select this rel-type, then we execute **Transform** / **Rel-type** / **-> Entity type**. The result is shown in Figure 9.13.

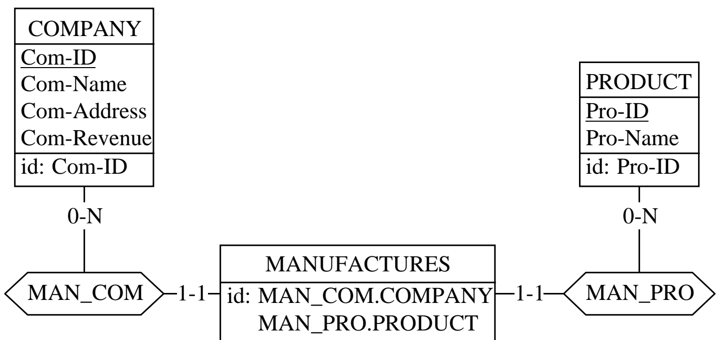


Figure 9.13 - Entity type equivalent of a *many-to-many* rel-type.

The rel-type has disappeared, and has been replaced by an entity type. The identifier of `MANUFACTURES` is worth being examined. In the source schema, there cannot be more than one relationship between a given company and a given product. Therefore, according to the resulting schema, there cannot exist more than one `MANUFACTURES` entity depending on the same `COMPANY` and the same `PRODUCT`. Hence the identifier.

Now, let us consider how we could further transform this schema into a pure SQL-compliant version. The task is now easy: it suffices to apply the *Rel-type to Reference attribute* transformation we just studied in this lesson. We get, quite naturally the schema of Figure 9.14.

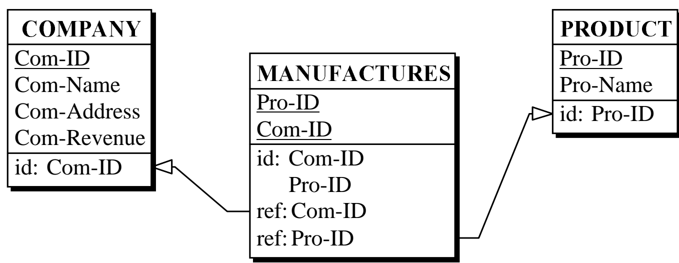


Figure 9.14 - The SQL-compliant expression of a *many-to-many* rel-type.

Why do we proceed in two steps? First, as we have proved it, this does not prevent us to reduce *many-to-many* rel-types into pure relational structures easily. Then, the *Rel-type to entity type* transformation can be used in many other situations, for instance when building a conceptual schema[3]. Finally, this transformation admits an interesting inverse operator: transforming a *rel-type into an entity type*. You can exercise it easily on the previous example.

9.7 Transforming complex relationship types

This transformation is more general than suggested in the previous section. In fact, it can be used to transform *any rel-type* into an equivalent entity type. For instance, it can be applied on:

- N-ary rel-types (with degree greater than 2),
- rel-types with attributes (N-ary, binary),
- one-to-many rel-types,
- one-to-one rel-types,
- cyclic rel-types.

To get an idea of these applications, we will transform an excerpt of the schema developed in Lesson 4, which includes a complex entity type which has 3 roles, an attribute and an explicit identifier! (Figure 9.15).

3. For instance, if a fact currently is represented by a rel-type, and a statement suggests to link it to an existing entity type, this rel-type should first be transformed into an entity type.

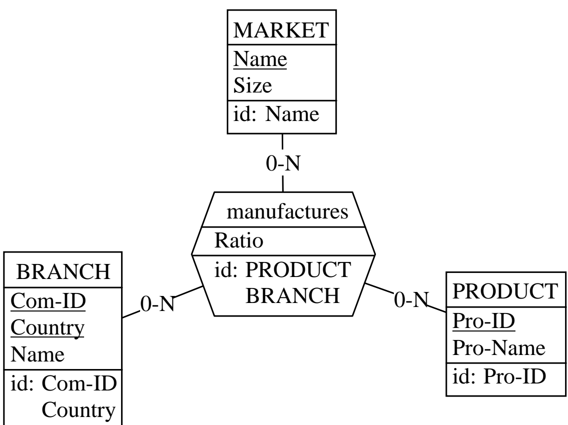


Figure 9.15 - A complex *3-ary* relationship type.

It can be transformed into the schema of Figure 9.16, which can, if needed, be further transformed into SQL-compliant structures.

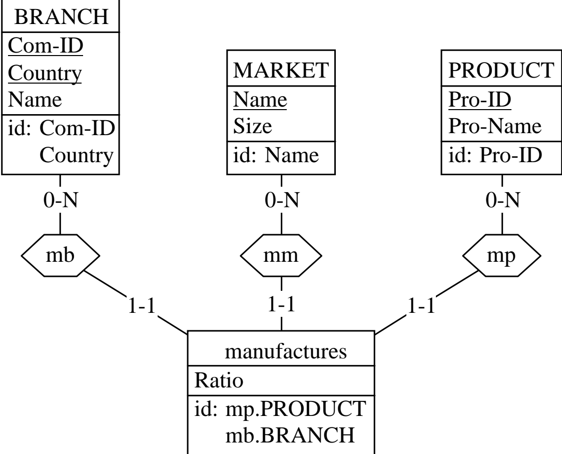


Figure 9.16 - Transformation of a complex 3-ary rel-type into an entity type.

9.8 Logical design, at last!

We will apply all this new knowledge on our conceptual schema. Of course, we lack some techniques, for instance to process multivalued attributes. But we know enough to reduce all the rel-types, which is a first move in the good direction.

First of all, we create a new schema by copying the conceptual schema. We select the latter, then we ask **Product** / **Copy product**. We give it the version name `Logical`. The project windows now contains the hierarchy of Figure 9.17.

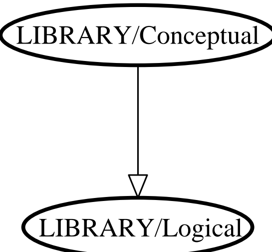


Figure 9.17 - The logical schema derives from the conceptual schema.

We open this logical schema (which so far is a mere copy of the conceptual schema), in order to transform it into a true SQL logical schema.

Processing the one-to-many rel-types

Let us tackle the easy problems first, namely the *one-to-many* rel-types. The most visible is rel-type `of` between `BOOK` and `COPY`, which can be processed as in Figure 9.18.

The next one, `responsible`, is a bit special: it is a *cyclic* rel-type. Nevertheless, the transformation works as usual: it includes in `BORROWER` a foreign key to the target entity type, that is, `BORROWER` itself (Figure 9.19).

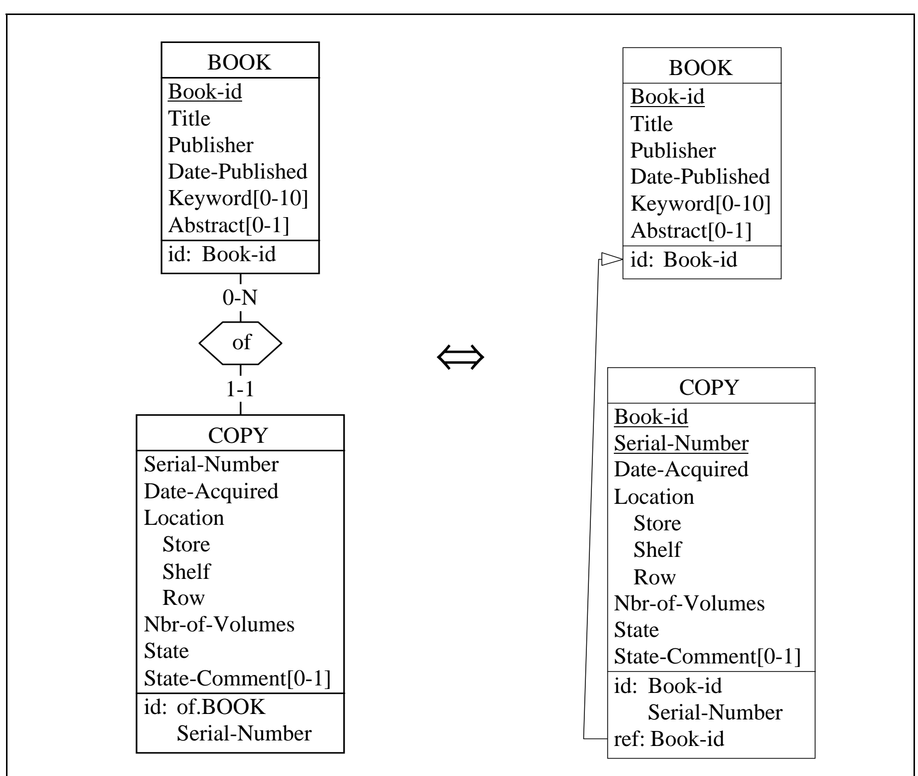


Figure 9.18 - Processing of the *one-to-many* rel-type `of`.

Processing the *many-to-many* rel-types

We can now address the other rel-types. We suggest to transform the *many-to-many* rel-types first.

`reference` is a cyclic *many-to-many* rel-type. It is first transformed into an entity type and two *one-to-many* rel-types as illustrated in Figure 9.20.

Then, these new rel-types are reduced to foreign keys (Figure 9.21).

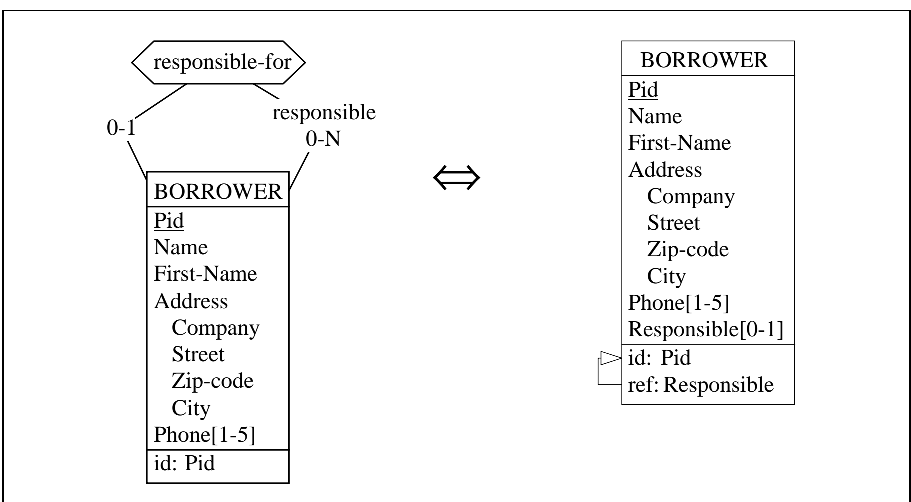


Figure 9.19 - Processing of the cyclic *one-to-many* rel-type `responsible-for`.

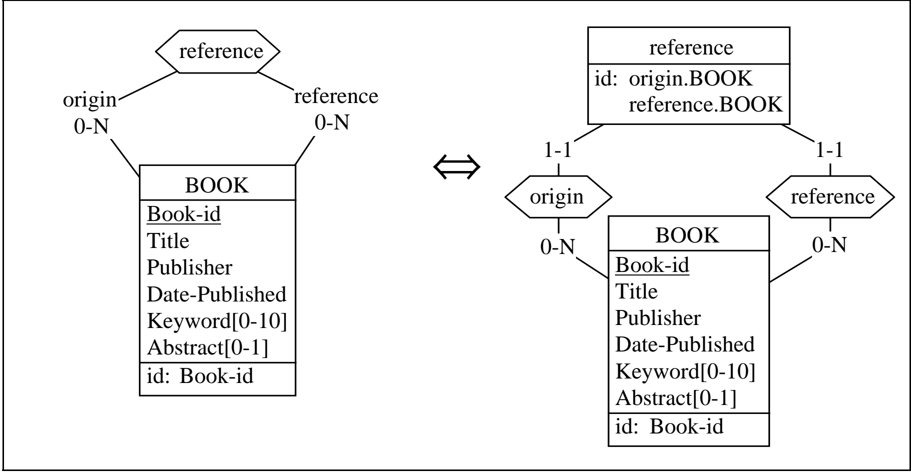


Figure 9.20 - Reducing a cyclic *many-to-many* rel-type - Step 1.

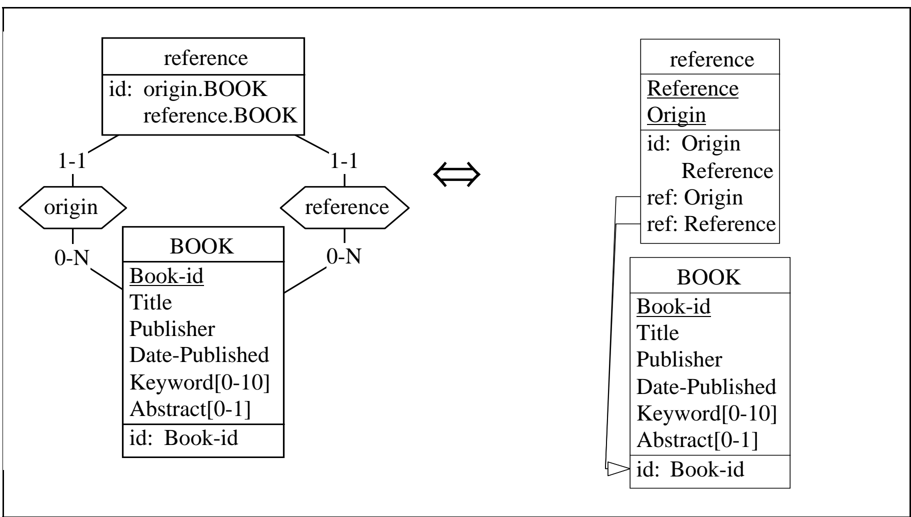


Figure 9.21 - Reducing a cyclic *many-to-many* rel-type - Step 2.

Then we transform the *many-to-many* rel-type `written`. We first replace it by an entity type (Figure 9.22). The rel-types may have strange names, but we do not care, because they are to be replaced.

Replacing `wri_BOO` is no problem. It is transformed into a foreign key (Figure 9.23).

However, when we ask DB-MAIN to transform `wri_AUT` in same way, it refuses!! "*the referenced entity type must have an all-attribute identifier*" it says. Unfortunately, it is right: `wri_AUT` cannot be replaced by a foreign key if `AUTHOR` has no (primary) identifier.

To get rid of this rel-type, we have no other means but adding an identifier to `AUTHOR`. DB-MAIN knows this problem, and if we have no imagination, it can help us by adding to this entity type a technical identifier. Let us try this: we execute the command **Transform** / **Entity type** / **Add Tech ID** (we can change the default name, type and length of this new attribute). The result is in Figure 9.24.

`AUTHOR` is ready to be referenced by a foreign key. So we ask again the transformation of `WRI_AUT`, which now succeeds (Figure 9.25).

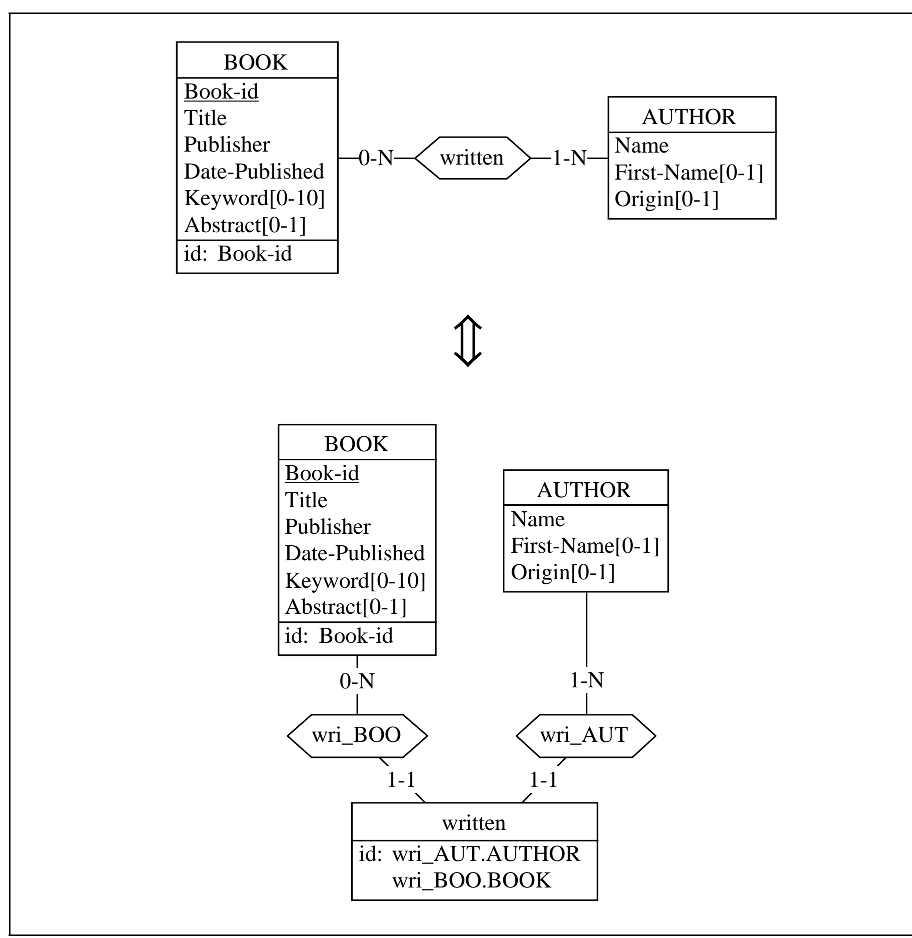


Figure 9.22 - Reducing the *many-to-many* rel-type `written` - Step 1.

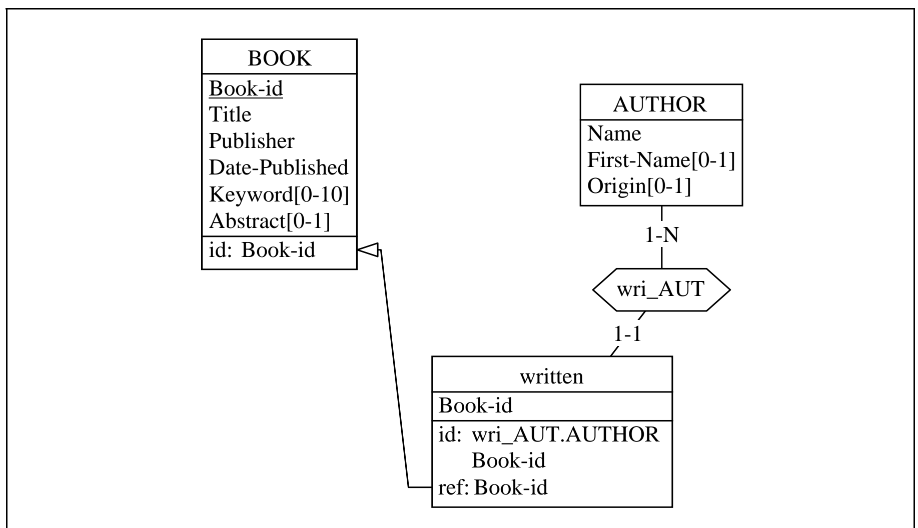


Figure 9.23 - Reducing the *many-to-many* rel-type `written` - Step 2.

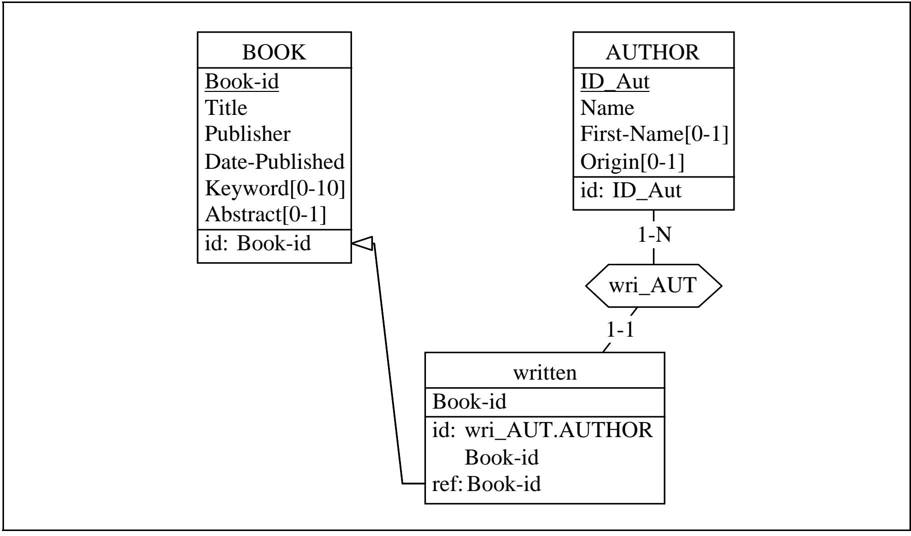


Figure 9.24 - `wri_AUT` can only be reduced when `AUTHOR` is given an identifier.

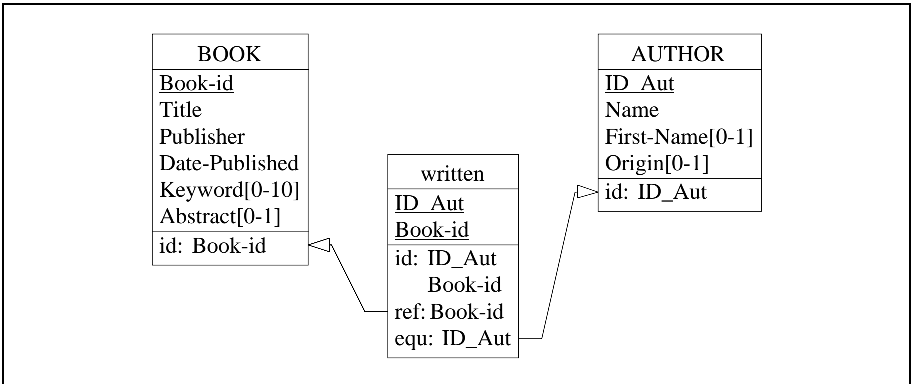


Figure 9.25 - The complete SQL-compliant expression of rel-type written.

Note the special form of the foreign key (ID_Aut) to AUTHOR: it has been specified as equ, and not as ref. This means that each ID_Aut value in written must be an ID_Aut value in AUTHOR, and conversely (hence the name equ*ality*). This constraint has been described in Lesson 5, but now, we can relate it to its origin: the [1-N] cardinality of the former rel-type WRITTEN.

Processing the N-ary rel-types

Let us now process the two N-ary rel-types. The first one is borrowing (Figure 9.26).

borrowing is first transformed into an entity type (Figure 9.27).

Note that bor_COP is *one-to-one*, due to the [0-**1**] cardinality of the role it derives from.

As usual, observing the identifiers is an infinite source of intellectual joy. In this case the surprising fact is the absence of identifier of entity type borrowing. Does it mean that this entity type is similar to AUTHOR. Not quite. In fact, this entity type has a sort of *implicit identifier*, which is COPY itself. Indeed, since bor_COP is *one-to-one*, if you designate a COPY entity, you cannot get more than one associated borrowing entity.

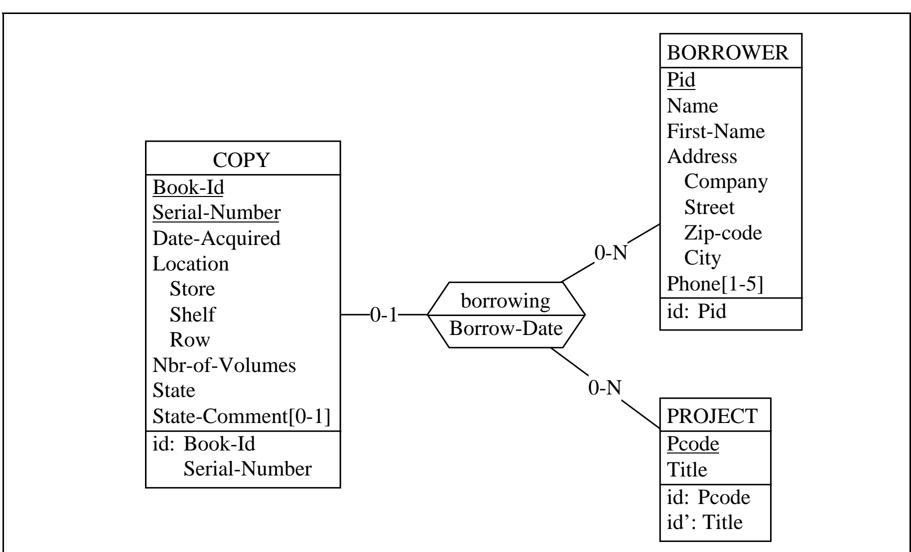


Figure 9.26 - The complex rel-type `borrowing`.

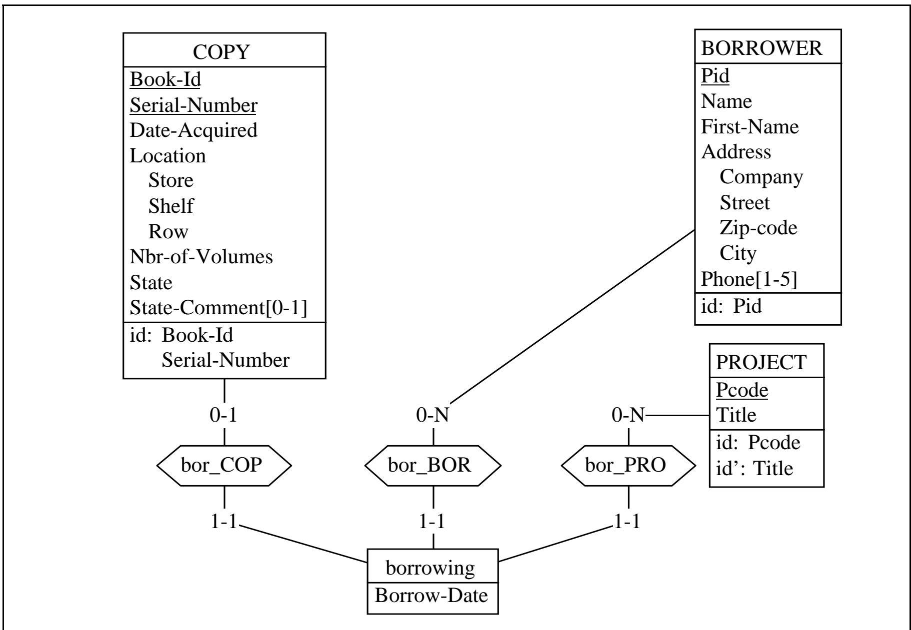


Figure 9.27 - Reducing the complex rel-type `borrowing` - Step 1.

Reducing each rel-type to a foreign key gives us the result of Figure 9.28. We observe that the identifier of `borrowing` has been made explicit: (`Book-ID`, `Serial-Number`) is both a foreign key and an identifier.

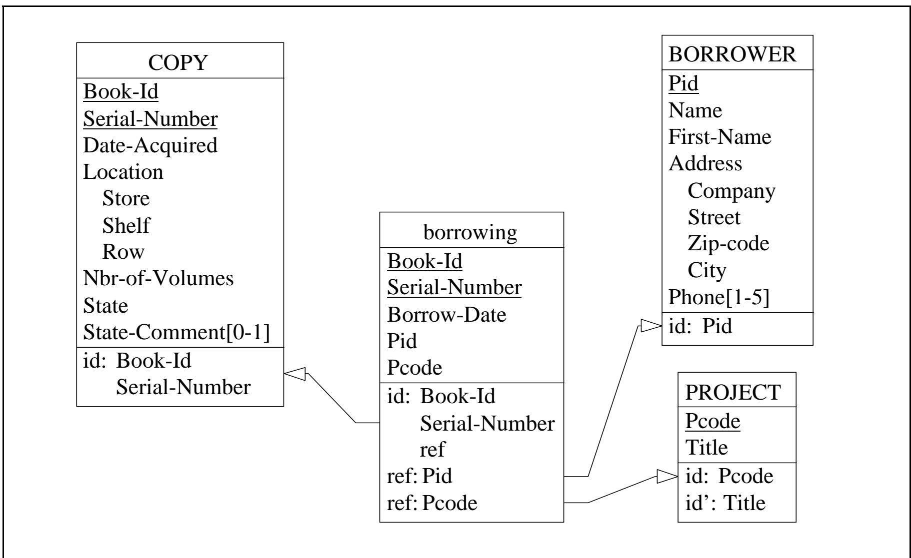


Figure 9.28 - Reducing the complex rel-type `borrowing` - Step 2.

The same procedure can be applied to `closed-borrowing` (Figure 9.29) It leads to the scenario of Figure 9.30 and Figure 9.31.

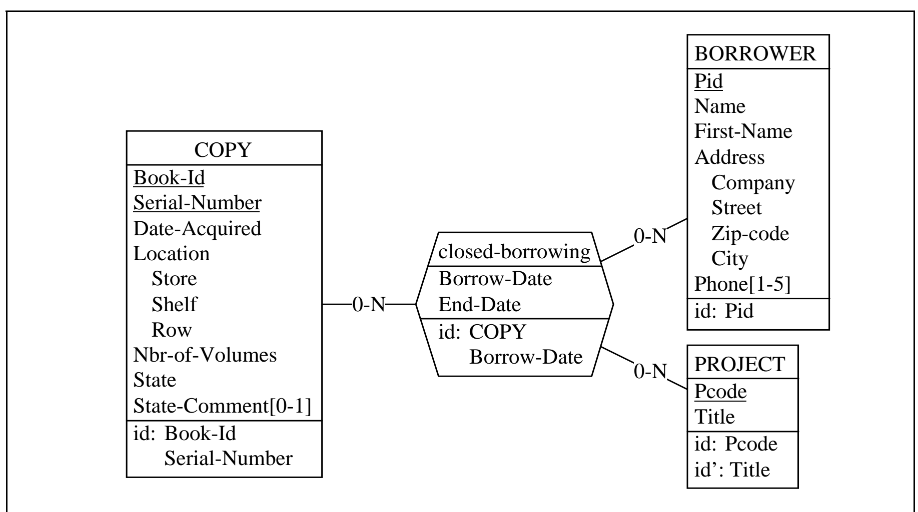


Figure 9.29 - The complex rel-type `closed-borrowing`.

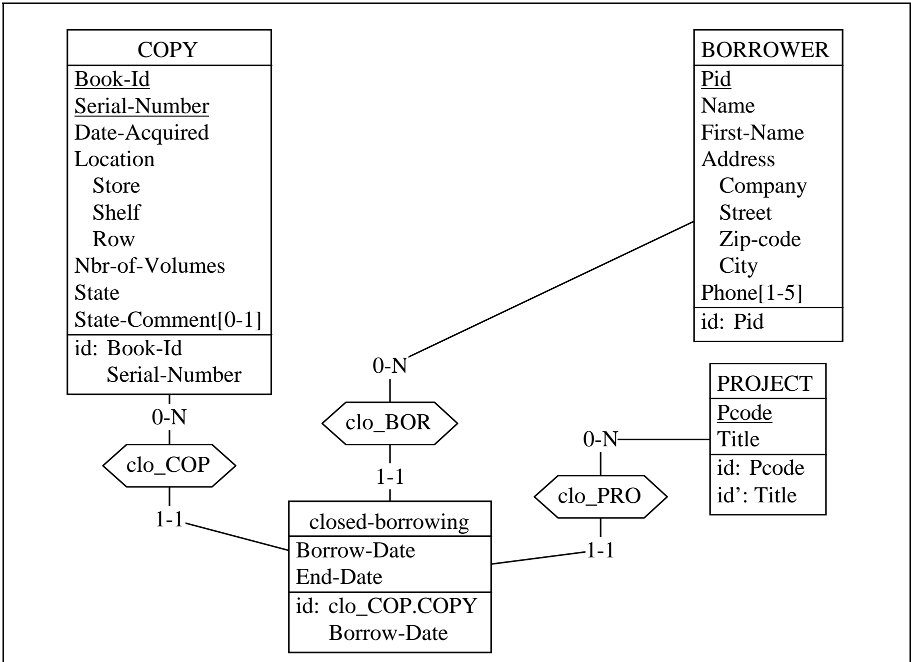


Figure 9.30 - Reducing the complex rel-type `closed-borrowing` - Step 1.

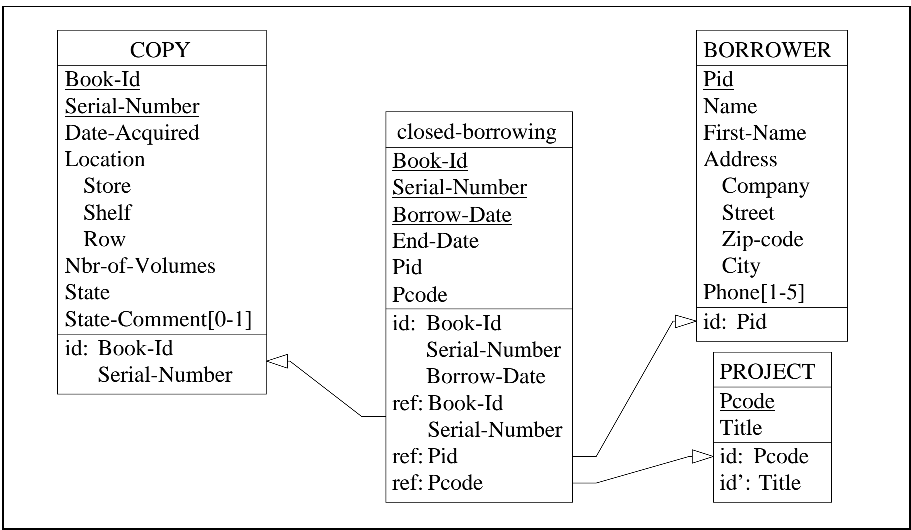


Figure 9.31 - Reducing the complex rel-type `closed-borrowing` - Step 2.

If everything worked correctly, there is no rel-types any more. Our schema should look like that of Figure 9.32.

Of course, this schema is not fully SQL-compliant yet. For instance, it includes compound and multivalued attributes which should be further processed. But it's late now, and we should better leave this task to another lesson.

9.9 Quitting the lesson

We save the current project under the name `logical-9.lun` and we quit DB-MAIN.

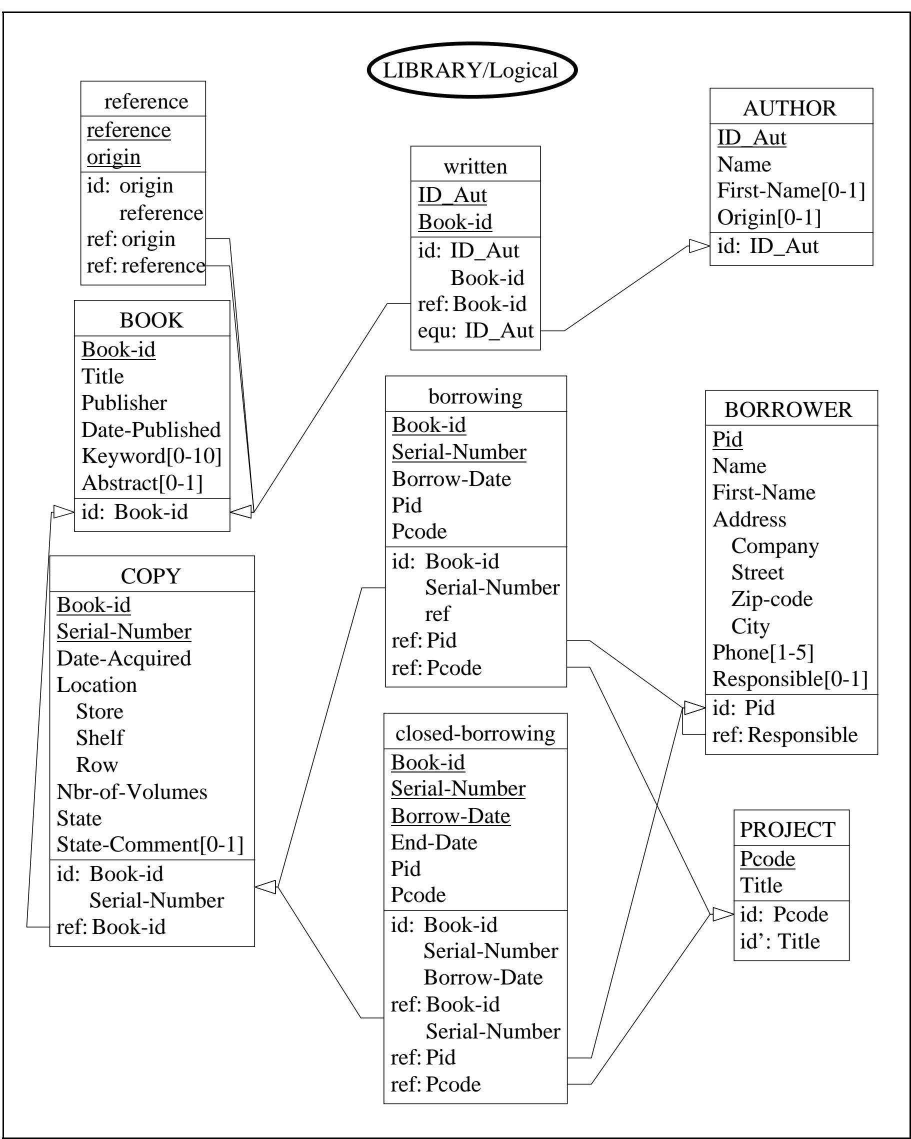


Figure 9.32 - The final schema - First version.

Technical addenda

9.10 On the *rel-type/entity type* transformation

This transformation is an important step torwards fully relational schemas. It can be used in many other contexts too, such as conceptual design, schema optimization or logical design of other kinds of databases such as object-oriented databases and standard files for instance.

It also induces an interesting property on the equivalence of schemas. There have been (and still are) warm discussions on the best representation of some concepts or facts: should they be represented by relationship types or by entity types? The answer often is: don't care, because both representations are proven to be equivalent, and therefore are both valid. For instance, the schemas of Figure 9.33 are strictly equivalent, and choosing one of them is not worth being disputed very long[4].

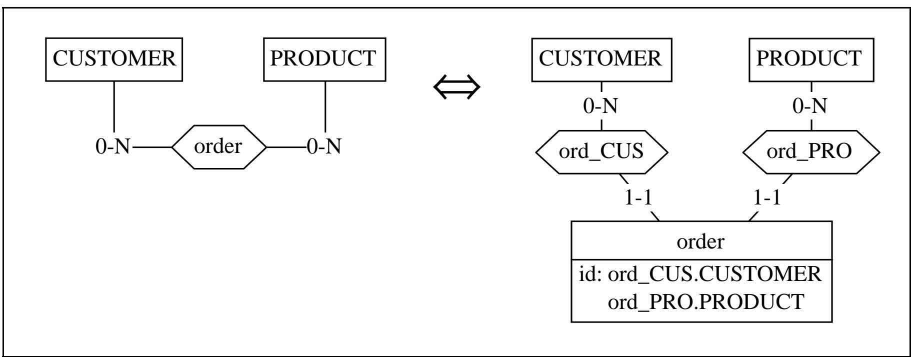


Figure 9.33 - The fact that *customers order products* can be represented by a rel-type as well as by an entity type.

Anyway, this transformation needs some additional discussion on how the semantics of the rel-type can be propagated to its equivalent entity type.

First of all, if we consider the populations of the entity types (i.e., sets of entities) and of the rel-types (i.e., sets of relationships[5]), we have to understand

4. As a matter of fact, both can be translated into the same SQL representation.

that each `order` relationship in the left-side schema is represented by an `ORDER` entity in the right-side schema. Moreover, we also have to observe that each role in the left-side schema is transformed into a *one-to-many* (or sometimes *one-to-one*) rel-type in the right-side schema.

This being said, we will precise a little bit the three basic propagation rules of this transformation (Figure 9.34).

1. *Cardinality propagation*

   For each (left) role `r` (with cardinality [i-j]) in rel-type `R`, there is a new (right) rel-type `r` with cardinalities ([i-j],[1-1]); this rule is illustrated by the pattern of Figure 9.34.

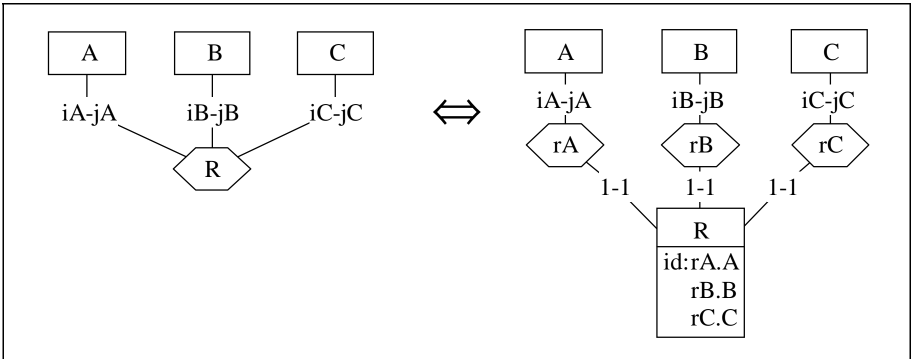


Figure 9.34 - Pattern of *rel-type/entity type* transformation.

2. *Attribute propagation*

   The attributes of (left) rel-type `R` are associated with (right) entity type `R`; this rule is illustrated by the transformation of the 3-ary rel-type `manufacture` in Figure 9.16.

3. *Identifier propagation*

   The identifiers of (left) rel-type `R` are translated for (right) entity type `R`: each role is replaced with the corresponding role of the new rel-type, and

5. A relationship can be visualized as an arc between the related entities.

each attribute is kept unchanged; this rule is illustrated by the transformation of the 3-ary rel-type `manufacture` in Figure 9.30.

To make sure we have understood this rather theoretical material, we will examine some representative applications.

The first one is the transformation of a 3-ary rel-type with non standard cardinalities. *Observations*: propagation of the cardinalities; `DETAIL` has no explicit identifiers, but has an implicit one (which one?).

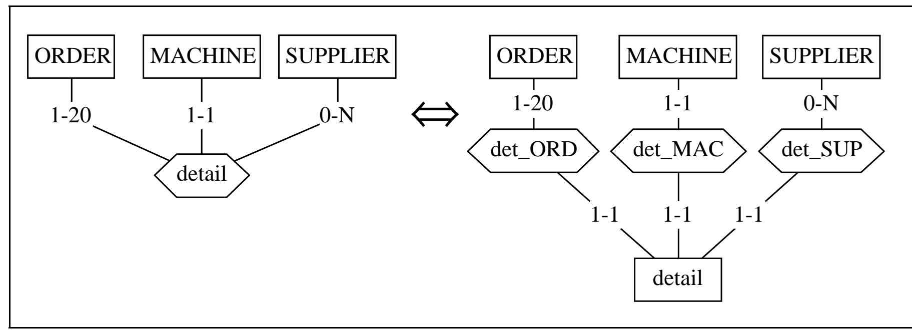


Figure 9.35 - Illustration of cardinality propagation.

The second application concerns a mere *one-to-one* rel-type (Figure 9.36). *Observations*: propagation of the cardinalities, which leads to two *one-to-one* rel-types; `of` has no explicit identifiers, has it an implicit one? How many implicit identifiers in fact?

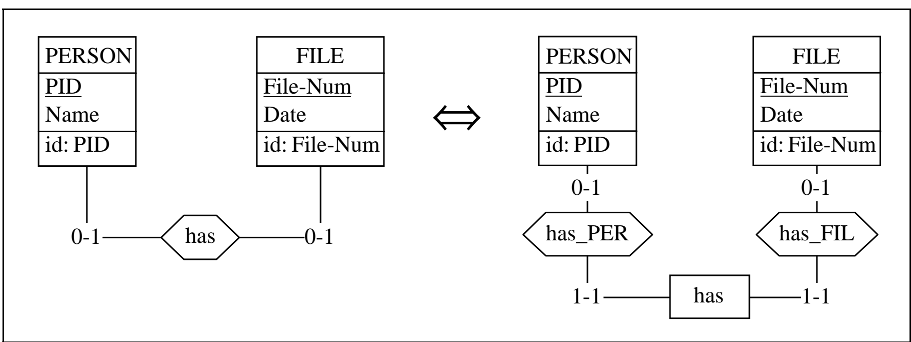


Figure 9.36 - Expressing a one-to-one rel-type as an entity type.

The third example is about *cyclic* rel-types (Figure 9.37). *Observations*: the cyclic structure has disappeared; `composed-of` has got an explicit identifier.

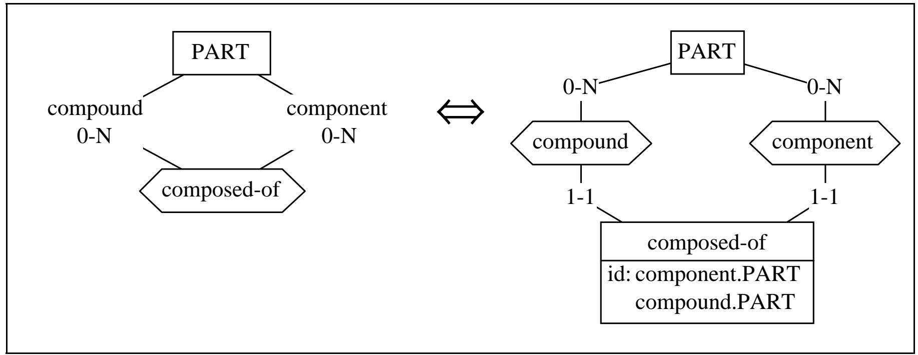


Figure 9.37 - Developping a *cyclic* rel-type into an entity type.

And the last one concerns *one-to-many* rel-types with attributes (Figure 9.38). *Observations*: propagation of the cardinalities, which generates a *one-to-one* rel-type; `works-in` has no explicit identifier; has it an implicit one?

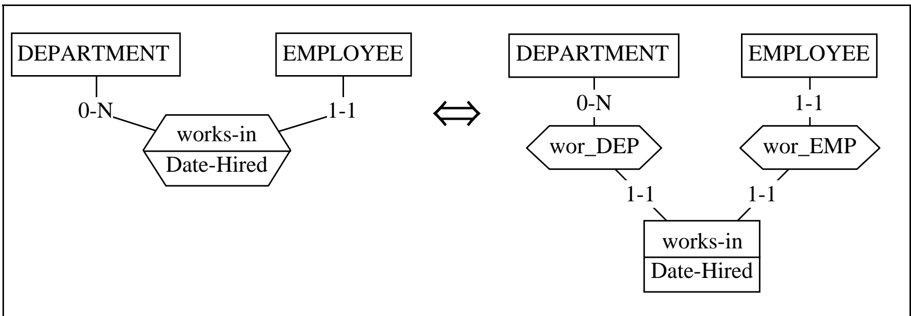


Figure 9.38 - Transforming a *one-to-one* rel-type with attributes

Reverse transformation: *entity type into rel-type*

As most transformation techniques we can use, this one is reversible. It means that when we encounter an entity type satisfying definite conditions, we can

transform it (or ask DB-MAIN to do it for us) into an entity type. We can call it the *Entity type to Rel-type* transformation.

What are these conditions on the entity type?

The first obvious condition is that it must take at least two roles. But it is not sufficient; for instance, these roles must be [1-1]. To be more precise, the entity type, say E, must satisfy each of the following conditions:

1. E plays at least two roles; *reason*: do try with only one!
2. the cardinalities of these roles must be [1-1]; *reason*: each E entity must be linked to one and only one entity of the each other sides; in the same way the (future) corresponding relationship will be made of one and only one entity of each kind;
3. all these roles belong to distinct rel-types; *reason*: otherwise one of the rel-type would be cyclic, and it would be a bit difficult to replace it by a role!
4. E has (at least) an implicit or an explicit identifier; *reason*: any two relationships of the same type are distinct, and cannot be made of the same entities and attribute values. Remember that any *one-to-one* rel-type makes an explicit identifier for each of its entity types.

9.11 On the *rel-type/reference attribute* transformation

This transformation is at the core of the logical design process for relational databases. In the version used in this lesson, this transformation generates *single-valued foreign keys* only. As you probably have observed, it can cope with multivalued foreign keys as well. Since such an extension would be useless to produce relational schemas[6], it will not be discussed in this lesson. However, the reverse transformation will address such structures, since they could be found in actual traditional data structures to be reverse engineered (recovering a conceptual schema from a logical schema).

With this restriction in mind, the conditions a rel-type `R` must satisfy to be replaced with a foreign key are easy to state. Let `R` be defined between entity types `B` and `A`.

1. `R` is *one-to-many*, or *one-to-one*; more precisely, the role of `B` is [0-1] or

6. On the contrary, it will be most useful to generate COBOL file structures or Object-oriented structures.

[1-1];

2. the other role (played by A) should be [0-1], [1-1], [0-N] or [1-N]; its cardinality can be different, but it will be considered [0-N] if it is [0-J], with J > 1, and as [1-N] if it is [I-J], with I > 0 and J > 1, and therefore translated incompletely[7];
3. A has at least one identifier; if one of them is primary, it must comprise attributes only; otherwise, there must exist at least one identifier made of attributes only.

Ideally, the foreign key of B should be a copy of a primary identifier of A, but the DB-MAIN tool can cope with more severe situations, where only secondary ids are available.

We will sketch the main variants of this transformation on a common framework (relations between *persons* and *files*), presented with different semantics.

The first situation concerns files which may describe persons (Figure 9.39). This is the most common case.

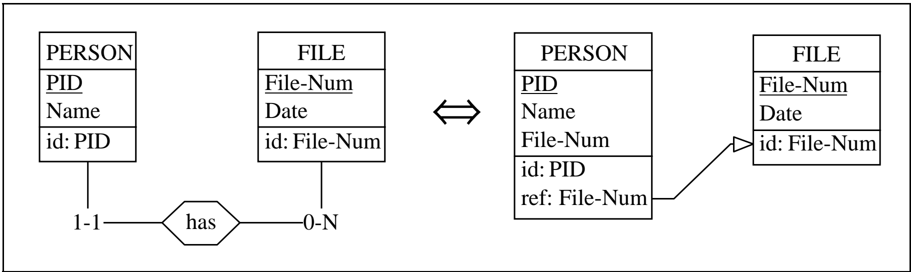


Figure 9.39 - Expressing a standard *one-to-many* rel-type as a foreign key.

Then we suppose that some persons are not described in any file. The foreign key becomes optional (Figure 9.40).

---

7. this means that, in such cases, the transformation is not fully semantics-preserving. Note however that DB-MAIN stores the non standard cardinality of the role in the description of the foreign key (see Figure 9.45, where this point is discussed). A more sophisticated SQL generator can therefore translate this cardinality into validation procedures.

If a file cannot describe more than one person, the foreign key becomes an identifier as well (Figure 9.41).

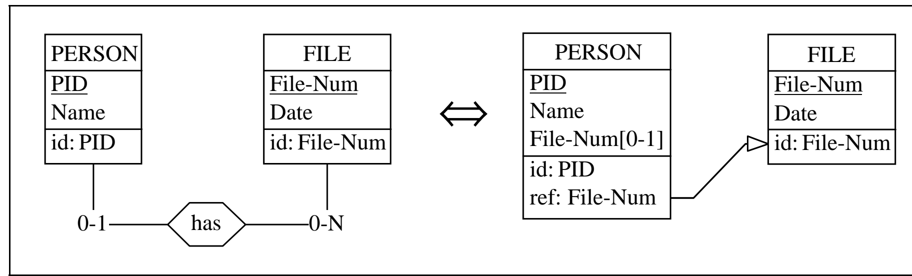


Figure 9.40 - Expressing an optional *one-to-many* rel-type as an optional foreign key.

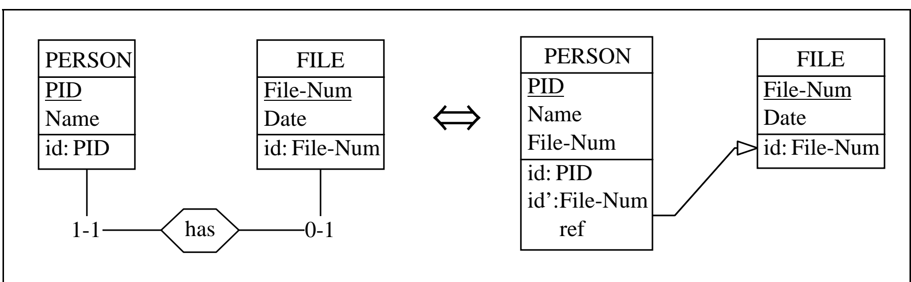


Figure 9.41 - Expressing a *one-to-one* rel-type as an identifying foreign key.

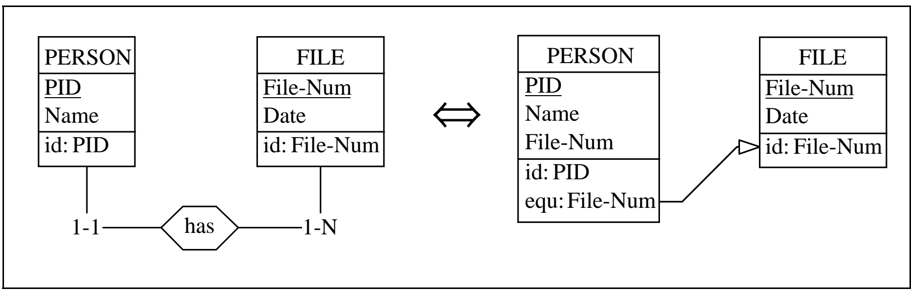


Figure 9.42 - Expressing a *bi-mandatory one-to-many* rel-type as an `equ` foreign key.

Now, let us suppose that each file describes at least one person. The referential constraint induced by the foreign key is complemented with an additional inclusion constraint, leading to an `equ`*ality* constraint (Figure 9.42).

If both roles are [1-1], the foreign key can be included in either side (Figure 9.43).

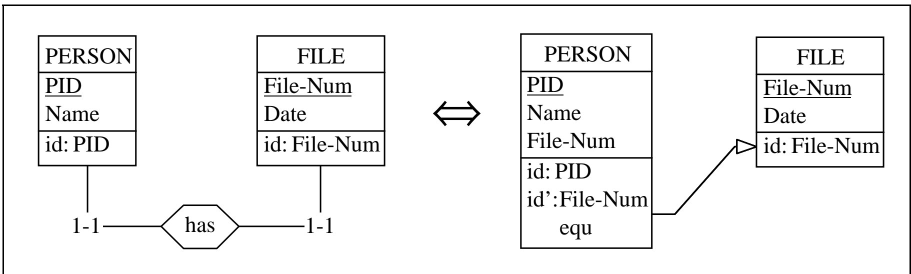


Figure 9.43 - Expressing a *bi-mandatory one-to-one* rel-type as an identifying `equ` foreign key.

Now, there are situations where the transformation, at least as it is generally implemented in CASE tools, does not preserve all the semantics of the source schema. That is the case for non standard values of the cardinalities[8]. Let us suppose that *a file describes up to 20 persons* (Figure 9.44)

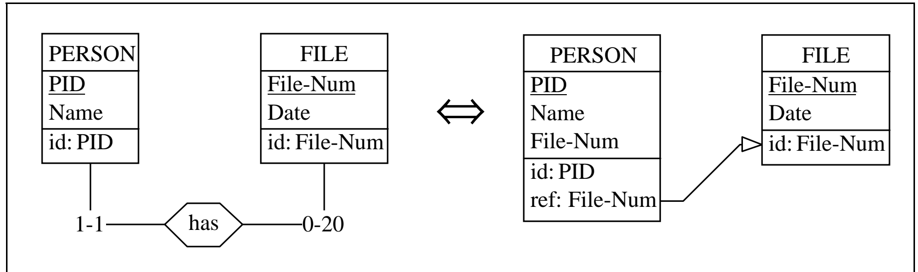


Figure 9.44 - Expression of a rel-type with non standard cardinalities. A loss of semantics may occur if the SQL generator does not produce the code that enforces this cardinality (through a trigger mechanism for instance).

8. I.e., other than [0-1], [1-1], [0-N], [1-N].

The cardinality [0-20] has been approximated by the more standard value [0-N]. In the same way [5-10] would have been transformed as if it were [1-N]. To prevent from this loss of semantics, the description of the foreign key now includes the exact cardinality (Figure 9.45). Generating the SQL code that enforces this constraint is up to the programmer or to the generator.

Figure 9.45 - The lost cardinality is stored in the description of the foreign key.

Let us now consider the situations where the referenced identifier is made up of several attributes. As expected, the foreign key is made of as many components as there are attributes in the identifier (Figure 9.46).

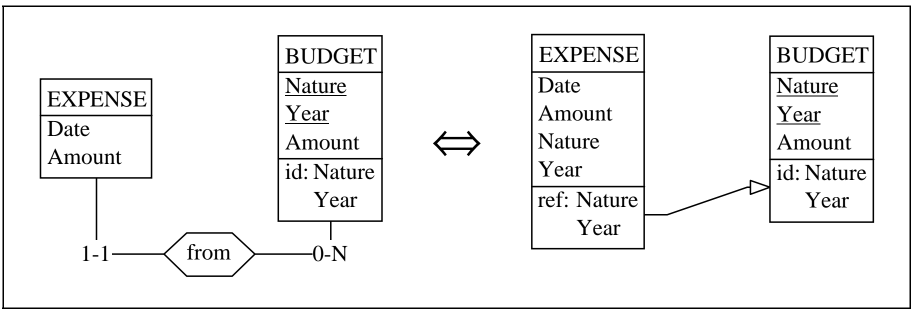


Figure 9.46 - A multi-component identifier in the target entity type induces a multi-component foreign key.

The transformation is more delicate if an expense can be made independently of a budget. In this case, the `from` rel-type is optional for `EXPENSE`, and the

foreign key must be optional as well. However, this optionality implies two properties:

- each component is **optional** (cardinality [0-1]),
- all the components form a **coexistence** group (Figure 9.47).

The latter constraint is important. Indeed, discarding it would allow an `EXPENSE` entity to have a value for `Nature`, and not for `Year`, a situation which cannot be represented in the left-side schema.

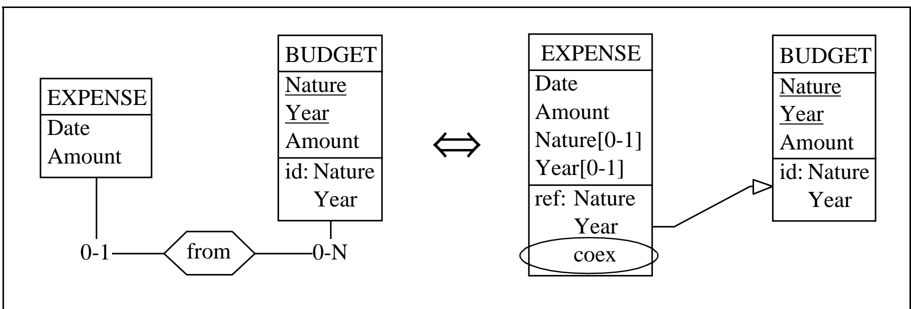


Figure 9.47 - An *optional multi-component* foreign key must be accompanied by a coexistence constraint.

Reverse transformation: *foreign key into rel-type*

As usual, this transformation can be interpreted the reverse way, as the transformation of *a foreign key into a rel-type*. All the situations presented in this addendum can be read from right to left. In other words, if a situation described in the right-hand side of one of these figures is encountered, it can be replaced by the corresponding schema in the left-hand side. For instance, the schema of Figure 9.48 can be interpreted as that of Figure 9.49.

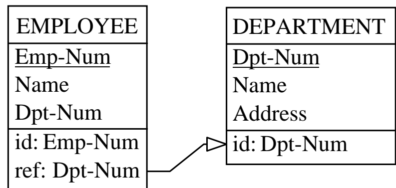


Figure 9.48 - An observed foreign key . . .

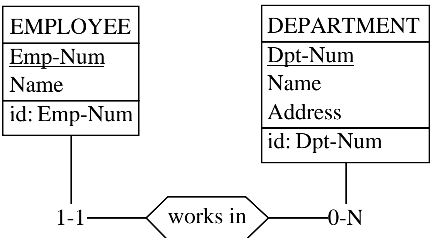


Figure 9.49 - . . . and its rel-type interpretation.

To carry out this transformation, we select the foreign key (through its group, not its attribute), then we execute the command **Transform** / **Group** / **-> Rel-type**.

Such transformations will be of prime importance in *reverse engineering* activities (see the book dedicated to this process).

Multivalued foreign keys

However, the DB-MAIN tool can address more complex foreign keys, such as those which can be encountered in, say, COBOL file structures. We will present an important extension, namely **multivalued foreign keys**.

A multivalued foreign key can reference more than one target entity. It is transformed into a *many-to-many* rel-type (Figure 9.50).

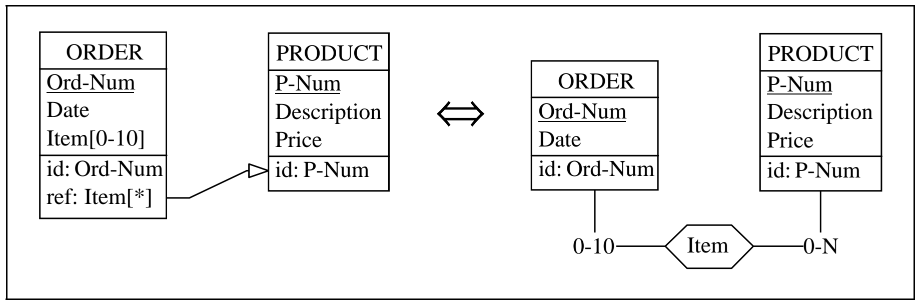


Figure 9.50 - A *non identifying multivalued* foreign key is interpreted as a *many-to-many* rel-type.

. . . or into a *one-to-many* rel-type if the foreign key is an idenfier as well (Figure 9.51). DB-MAIN can interpret multivalued foreign keys, but it can also generate them, though this function generally is of little interest for building relational databases.

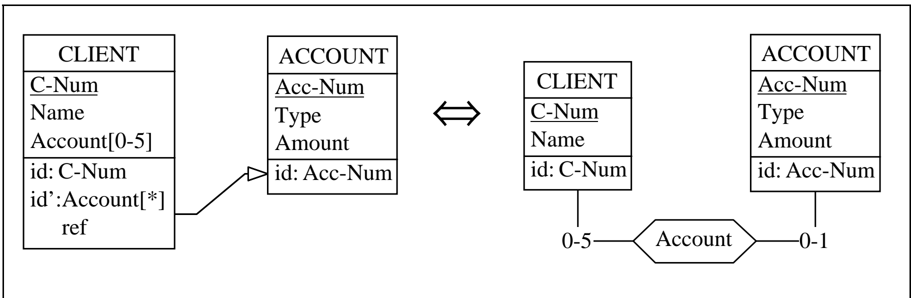


Figure 9.51 - An *identifying multivalued* foreign key is interpreted as a *one-to-many* rel-type.

9.12 On the *technical ID* transformation

We have been faced with a situation in which transforming a *one-to-many* rel-type was impossible due to the absence of identifier. We then used a specific transformation which, when applied to an entity type,

1. adds a technical attribute, and
2. makes it the primary identifier.

As it is now usual with transformations, this technique can have other useful applications. Let us examine its two main variants.

The first one introduces a primary identifier into an entity type which has none so far (Figure 9.52).

The second variant introduces a technical identifier into an entity type which already has one. In the example below, we suppose that `{Nature,Year}` is too complex, or too long, an identifier to be used as an identifier for `BUDGET`. For instance, this entity type should be referenced by a large number of foreign keys, leading to an important waste of space and poor performance. Introducing a short, meaningless, identifier will certainly help. This new attribute becomes the primary identifier, while the former one is given the secondary status (Figure 9.53).

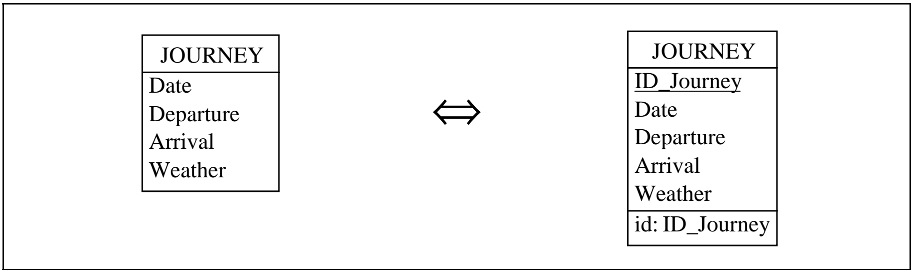


Figure 9.52 - Adding a technical identifier to an *unidentified* entity type.

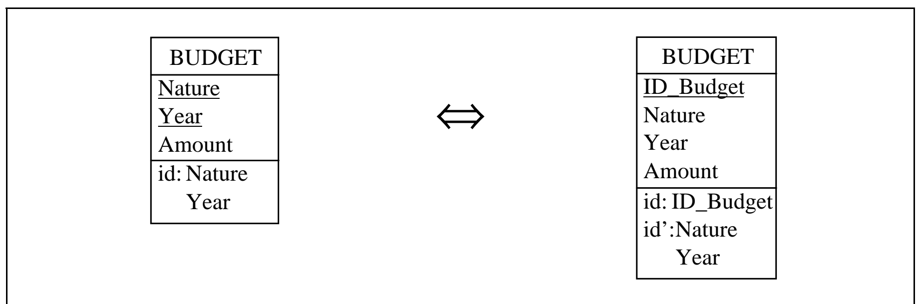


Figure 9.53 - Substiting a short technical identifier for a complex identifier.

The property of semantics preservation or reversibility is worth a comment. Indeed, as opposed to the other transformations encountered so far, this one does not replace one structure with another structure: it just adds some new structure.

Is it reversible? Yes indeed. The new structure bears absolutely no semantics (hence the term *technical id*). Therefore, introducing or removing it does not change the semantic content of the schema in any way. The transformation is trivially reversible.

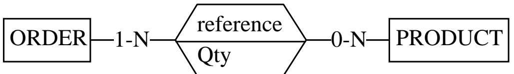
Summary of Lesson 9

- In this lesson, we have studied new notions:
  - logical design;
  - SQL-compliant schema;
  - schema equivalence;

- We have also learnt how
  - an Entity-relationship schema, called SQL-compliant, can represent relational database structures
  - to transform a rel-type into an entity type:
    Transform / **Rel-type** / **-> Entity type**
  - to transform an entity type into a rel-type:
    Transform / **Entity-type** / **-> Rel-type**
  - to transform a rel-type into reference attributes (foreign key):
    Transform / **Rel-type** / **-> Attributes**
  - to transform reference attributes (foreign key) into a rel-type:
    Transform / **Group** / **-> Rel-type**
  - to add a technical identifier to an entity type:
    Transform / **Entity type** / **Add Tech ID**

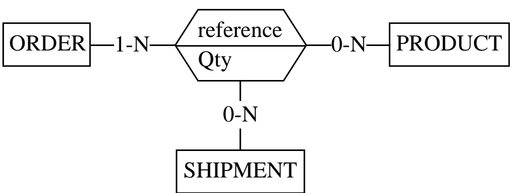
Exercises for Lesson 9

9.1 To represent the concepts underlying the sentence "*each order references one or several products, each in a given quantity* ...", a designer has chosen to propose the following schema:



Later on, he has to augment this schema by including the fact that " *shipments are made of one or several referenced products* ... ".

First, he proposes the solution below, obtained by adding a third role to the rel-type:

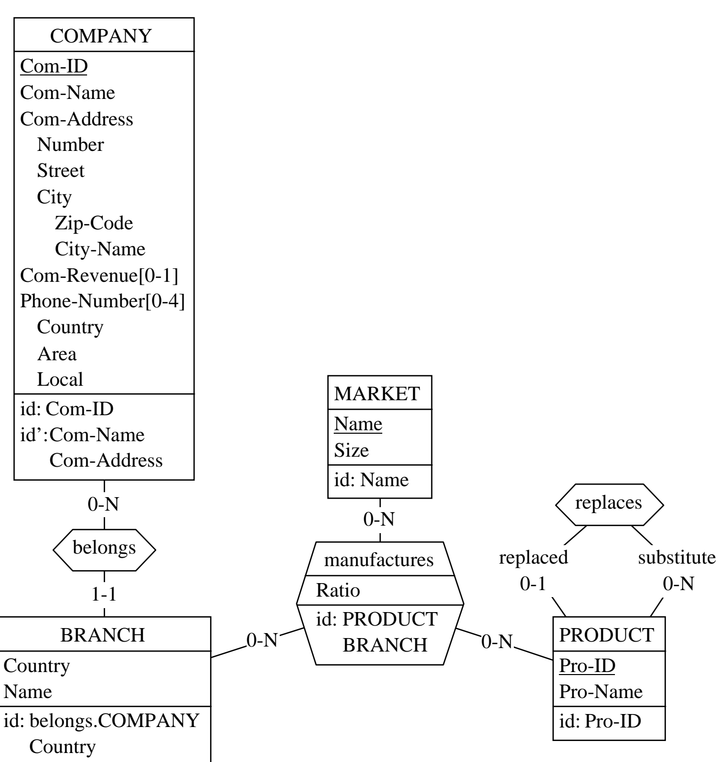


But finally, he does not feel quite satisfied and rejects it. He is right. Why? What correct solution could you propose?

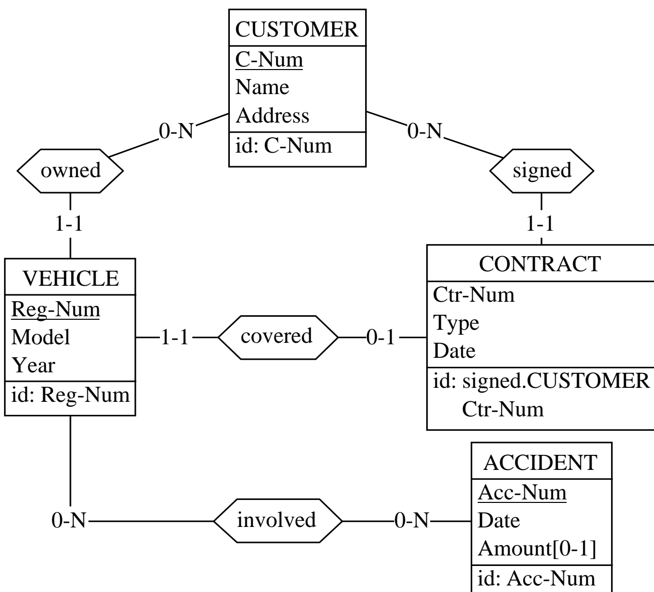
9.2 Propose an SQL-compliant schema for the following conceptual schema according to three procedures:

1. Enter this logical schema manually (a bit tedious but very instructive!)
2. Build it by applying selected transformations to the conceptual schema.
3. Ask the tool to do this job for you by **Transform** / **Relational model**.

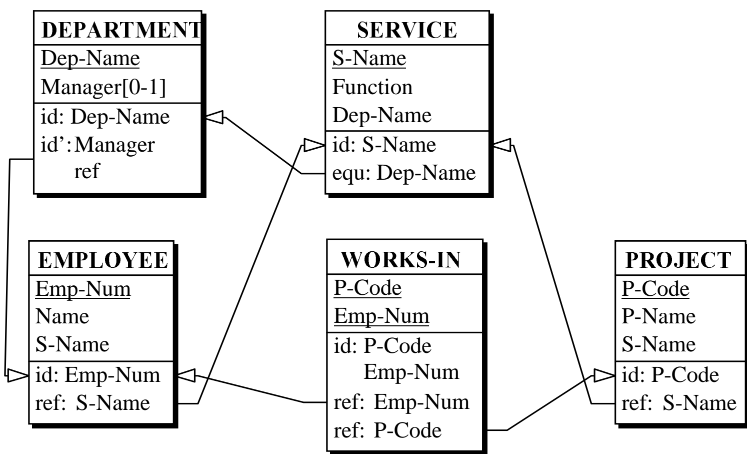
Compare the three solutions. Can you explain the differences?



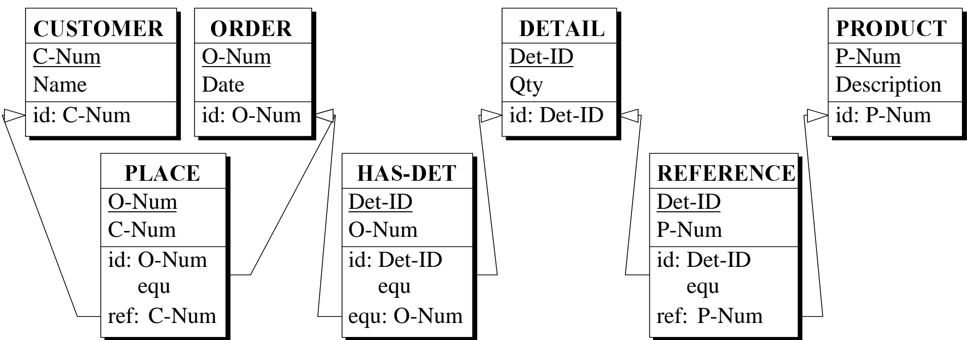
9.3 Propose an SQL logical database structure for the following conceptual schema.



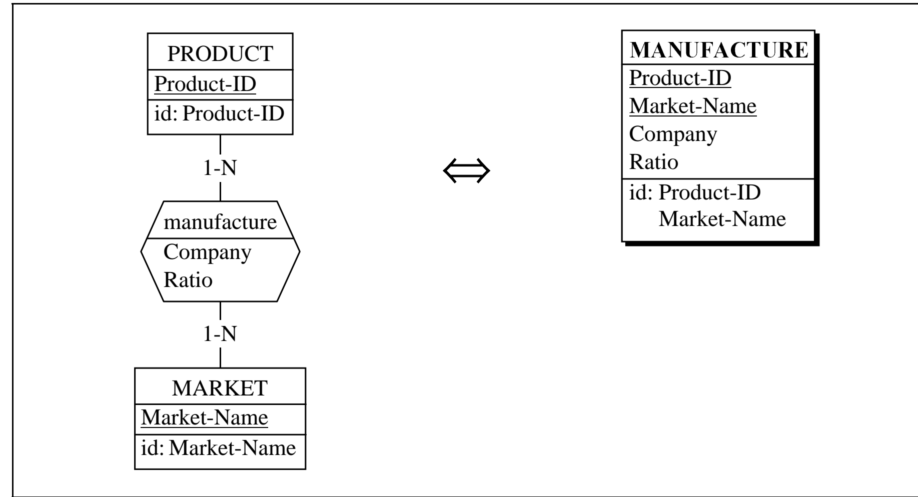
9.4 Retrieve in a systematic way the conceptual schema of this SQL logical schema (this problem is related to *Reverse engineering*).



9.5 Retrieve in a systematic way the most concise conceptual schema of this SQL logical schema.



9.6 In the Addendum of Lesson 8, we met the schema below (left). A designer proposes the schema on the right, claiming that it is equivalent to the former. What do you think of this.



Lesson 10

Logical Design (2)

Objective

We will complete the logical design of the LIBRARY database by processing compound and multivalued attributes. Then, we will discuss these transformations in greater detail.

10.1 Starting Lesson 10

We start DB-MAIN and we open the project logical-9.lun which now includes the conceptual schema of the database in project, as well as a first version of the logical schema, called LIBRARY/Logical. We save it as logical-10, the version on which we will work in this lesson.

10.2 What to do next?

We open the schema LIBRARY/Logical, and we examine it very carefully. Does it look like an SQL-compliant logical schema? Let us recall the main rules defining such schemas:

An **SQL-compliant** schema comprises only:
- entity types - single-valued and atomic attributes - identifiers - reference attributes

Thanks to the processing of Lesson 9, it has no relationship types any longer. All of them have been replaced by new entity types and by reference attributes (foreign keys). However, it still includes invalid attributes:

- Keyword (from BOOK) is multivalued,
- Location (from COPY) is compound,
- Address (from BORROWER) is compound,
- Phone (from BORROWER) is multivalued.

We will first tackle the compound attributes, then process the multivalued attributes.

10.3 Transforming the *compound* attributes

In lesson 6, when discussing the coexistence constraint, we examined a transformation (**Group** / **Aggregation**) which makes a compound attribute from a

group of individual attributes. We also mentioned another transformation, called *Disaggregation*, whose aim was precisely to undo the effect of the former, i.e., to replace a compound attribute with its components. We got it by the command **Transform** / **Attribute** / **Disaggregation**. That is just what we need here.

We select the attribute Location of COPY, and we execute this command. We then are asked what prefix we want to give to the names of the components. The tool proposes the prefix Loc_, which is the short name of the compound attribute, if any, or the first three characters of the name of this attribute. We can change it, or even delete it. We choose to accept the proposed prefix, and we click on button OK. The entity type COPY is transformed as in Figure 10.1.

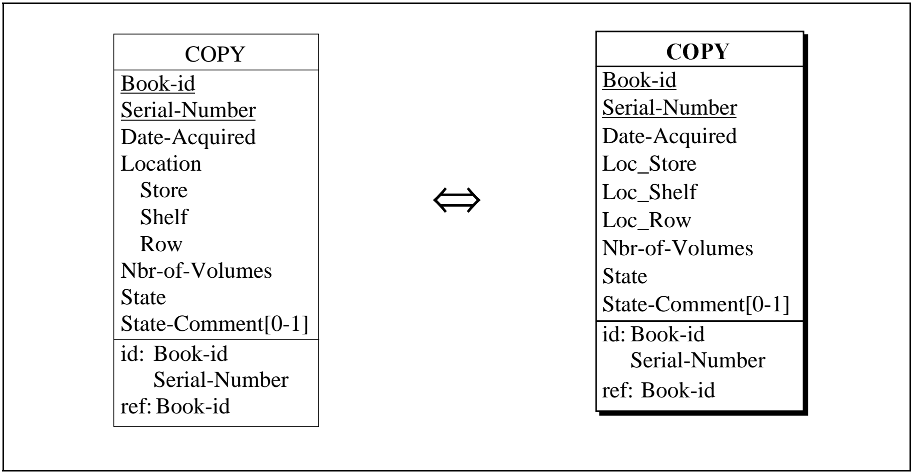


Figure 10.1 - Disaggregating the compound attribute Location.

Carefully choosing the prefix allows reminding the origin of these attributes. It is not a formal way to do so, but can be useful in many situations[1].

Processing the attribute Address of BORROWER follows the same procedure (Figure 10.2).

1. We will study in another volume how DB-MAIN can remember all the operations it has been asked to execute. The journaling functions, available through the Log menu, allows the recording and the replaying of selected operations.

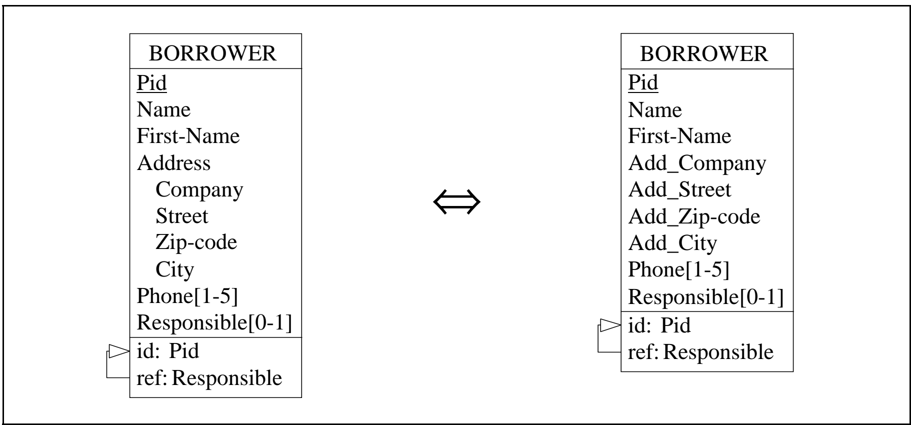


Figure 10.2 - Disaggregating the compound attribute `Address`.

10.4 Transforming the *multivalued* attributes

We have already processed multivalued attributes by replacing them with equivalent entity types. Remember: when we analyzed the concept of *copy of book* in lesson 7, we first represented it by the multivalued attribute `Serial-Number` of `BOOK`. Then, when we learned more about copies, we decided to represent them by a specific entity type. This entity type was derived from the attribute `Serial-Number` by transforming it into the entity type `COPY`. For this purpose, we used the transformation **Transform** / **Attribute** / **-> Entity type**.

This transformation produces a new entity type, which is quite SQL-compliant, but also a rel-type, which is not at all SQL-compliant! However, rel-types are no longer a problem for us, since we can get rid of any kind of rel-type, as we exercised it in lesson 9.

So, let us proceed as suggested: we select `Keyword` in `BOOK`, and we execute **Transform** / **Attribute** / **-> Entity type**. Now we are facing a puzzling question: do we prefer the *Instance representation* or the *Value representation* versions? We have discussed the differences between them, but you sure have forgotten it. Don't worry, this a good opportunity to have a deeper look at this important question.

First we choose the *Instance representation* procedure. The result is shown in Figure 10.3.

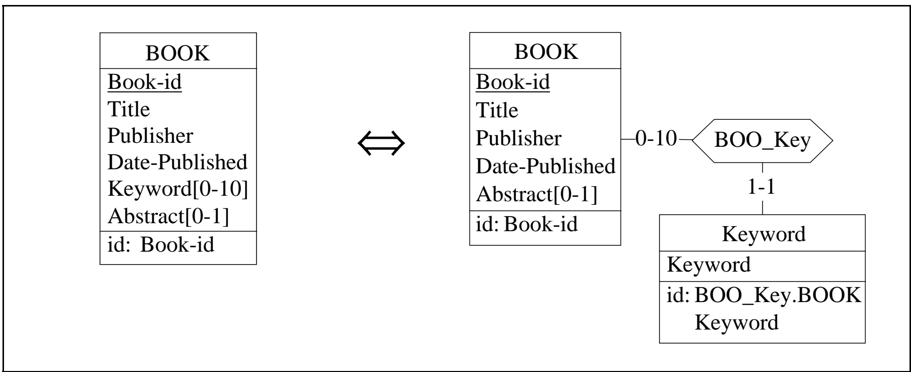


Figure 10.3 - Extracting the attribute `Keyword` as a new entity type through the *Instance representation* technique.

We then transform the rel-type `BOO_Key` into a foreign key as in Figure 10.4.

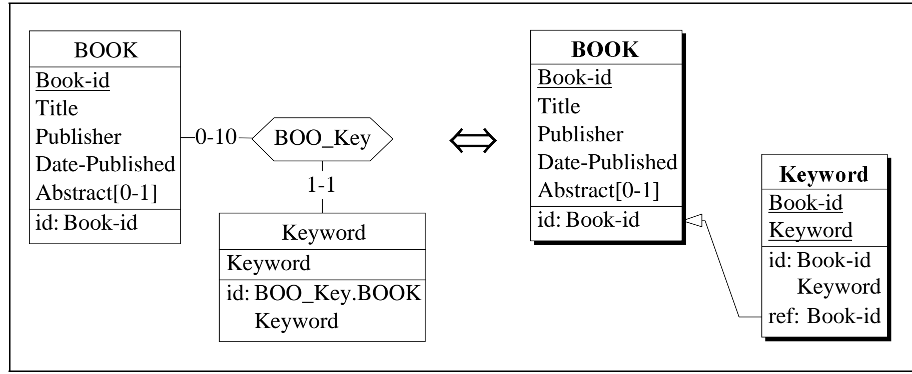


Figure 10.4 - Reducing the rel-type `BOO_Key` into a foreign key.

We must however be aware of a semantic loss which is induced by this transformation. Indeed, as discussed in the addenda of Lesson 9, this transformation does not translate completely non standard cardinalities. As a consequence, the cardinality [0-10] of `BOOK` has been translated as if it were [0-N]. If this limit is essential, then we can use the advanced SQL generator

of DB-MAIN that can generate *triggers* or *check* predicates to validate this kind of constraint in real time if needed. To keep the discussion as simple as possible, we will ignore this processor in this volume.

It is interesting to observe what would happen should we have chosen the other variant of the transformation, i.e., *value representation*. Let us try it in Figure 10.5.

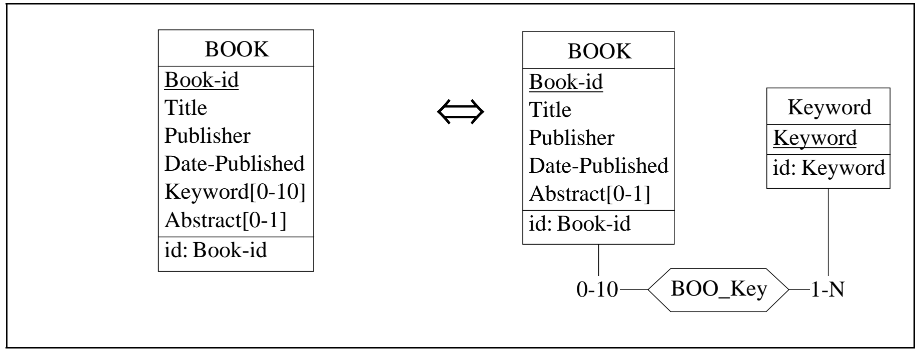


Figure 10.5 - Extracting the attribute `Keyword` as a new entity type through the *Value representation* technique.

Now, the rel-type `BOO_Key` is **many-to-many**. As in Lesson 9, we transform it into an entity type (Figure 10.6).

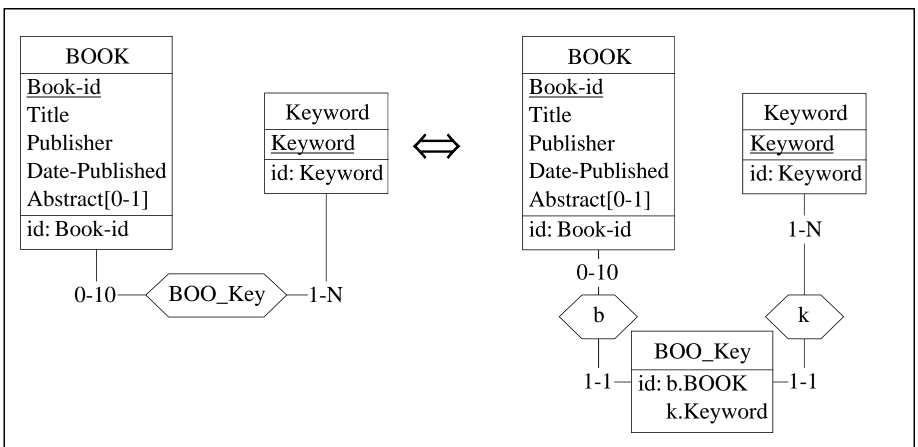


Figure 10.6 - Transforming the rel-type `BOO_Key` into an entity type.

Then we reduce each resulting *one-to-many* rel-type into a foreign key (Figure 10.7).

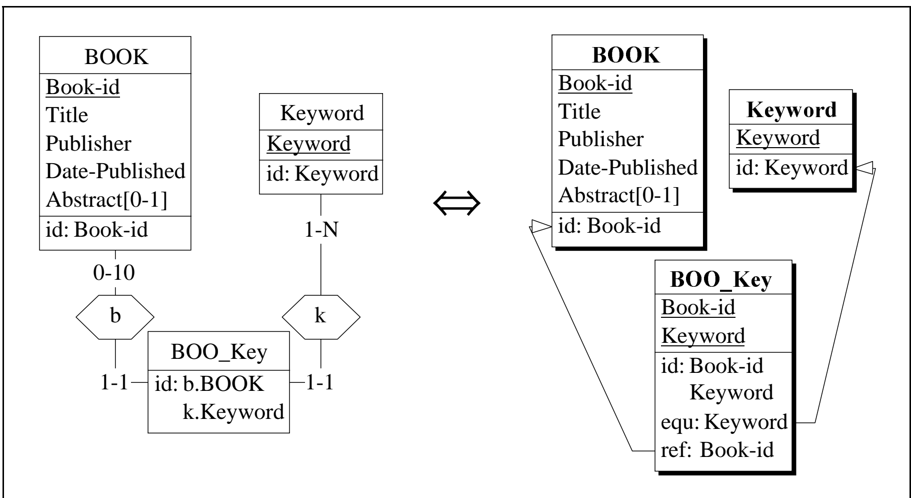


Figure 10.7 - Net result of extracting the attribute Keyword through the *Value representation* technique.

Now, how can we compare these solutions? Let us call them respectively the *Instance solution* (Figure 10.4) and *Value solution* (Figure 10.7).

A first observation is that the *Value solution* proposes two additional entity types while the *Instance solution* proposes one only. Therefore the latter will generate fewer SQL tables.

A second observation is that they are equivalent, i.e., they convey exactly the same semantics. Any situation in the application domain (the library) which can be represented by one of them can be represented by the other as well. Indeed, both solution derives from the same source schema, through semantics-preserving transformations[2]. In the Addendum, we will prove this equivalence in another way. To allow an easier comparison, we will present both solution side by side and change the names in such a way that a name has the same meaning in both schemas:

2. It is not quite true of course, because we have lost cardinality 10. However, since this property has been lost in both solutions, they still are equivalent.

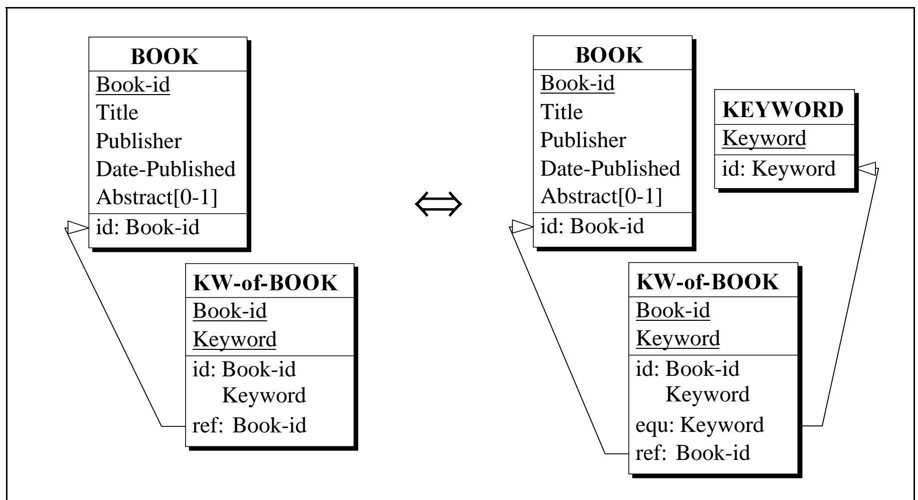


Figure 10.8 - Comparing both *Instance* (left) and *Value* (right) solutions.

Now, it is clear that the entity type KEYWORD represents the dictionary of all the **distinct values** of keywords in the library. This explicit representation was not required in the conceptual schema, and we will not give it a particular interest. Therefore, since we have no other criteria, we will choose the most economical proposal, i.e., the *Instance* solution.

The procedure to use to express the second multivalued attribute, Phone of BORROWER, will be exactly the same. To avoid confusing the new entity type with the representation of distinct phone numbers, we rename it Phone-of-Bor (Figure 10.9).

Here too, we lost the exact cardinality of Phone[1-5], replaced by [1-N].

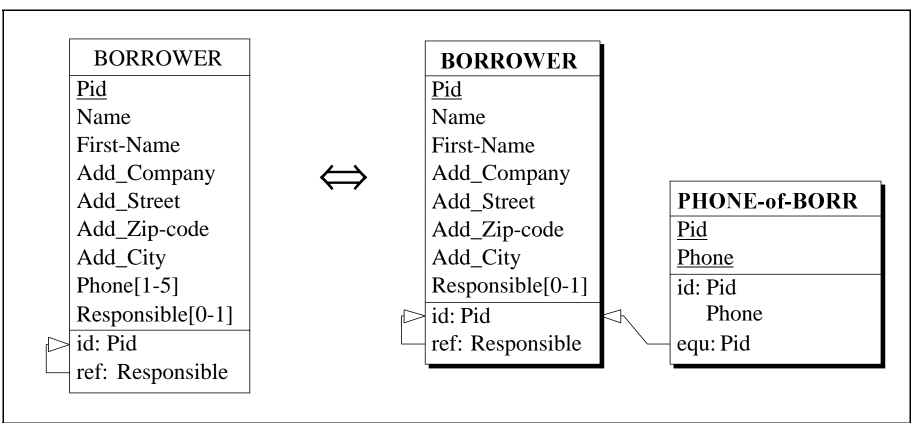


Figure 10.9 - Extracting the multivalued attribute `Phone` as a new entity type.

10.5 An (almost) SQL-compliant schema

Now we have completed the transformation of the conceptual schema into an SQL-compliant schema. The latter comprises only entity types, single-valued and atomic attributes, identifiers and foreign keys, and therefore can be expressed into SQL statements very easily, as will be shown in lesson 12. It is shown in Figure 10.10.

There is just a little point which deserves a comment. We have retained a constraint which does not seem to be SQL-compliant, namely the `equ` constraint, which is a special form of referential constraint that can be found in `AUTHOR` and in `BORROWER`. SQL can take care of the referential constraint, but will ignore the inverse inclusion constraint. So, it is able to implement the `ref` part of this constraint.

As a note for database programmers, we can suggest to express the full constraint as follows:

- the `ref` constraint will be expressed by a foreign key,
- the inverse *inclusion* constraint will be expressed as a `check` or `trigger` clause.
- the whole package will be encapsulated into *transactions* with deferred constraint checking.

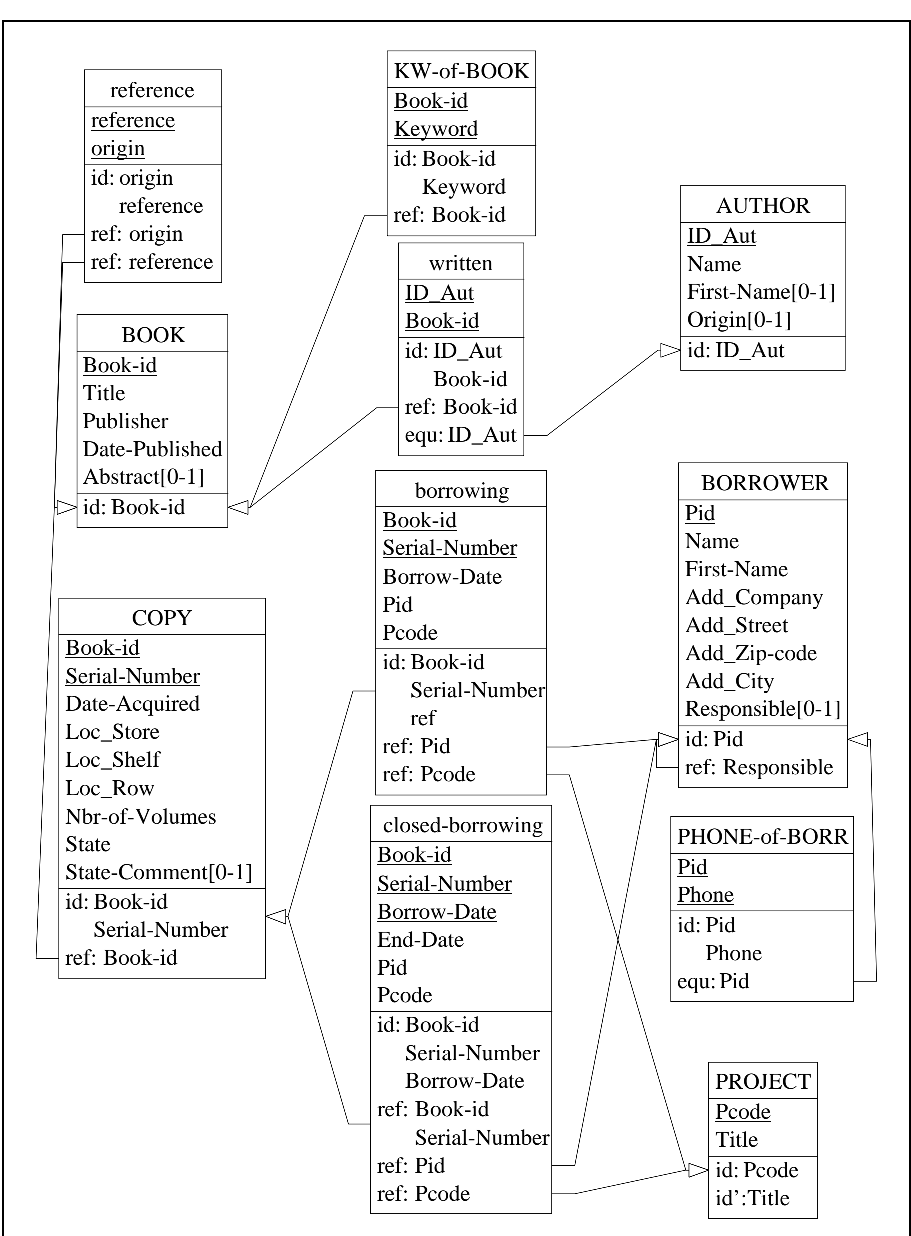


Figure 10.10 - This schema does not include non-relational constructs anymore.

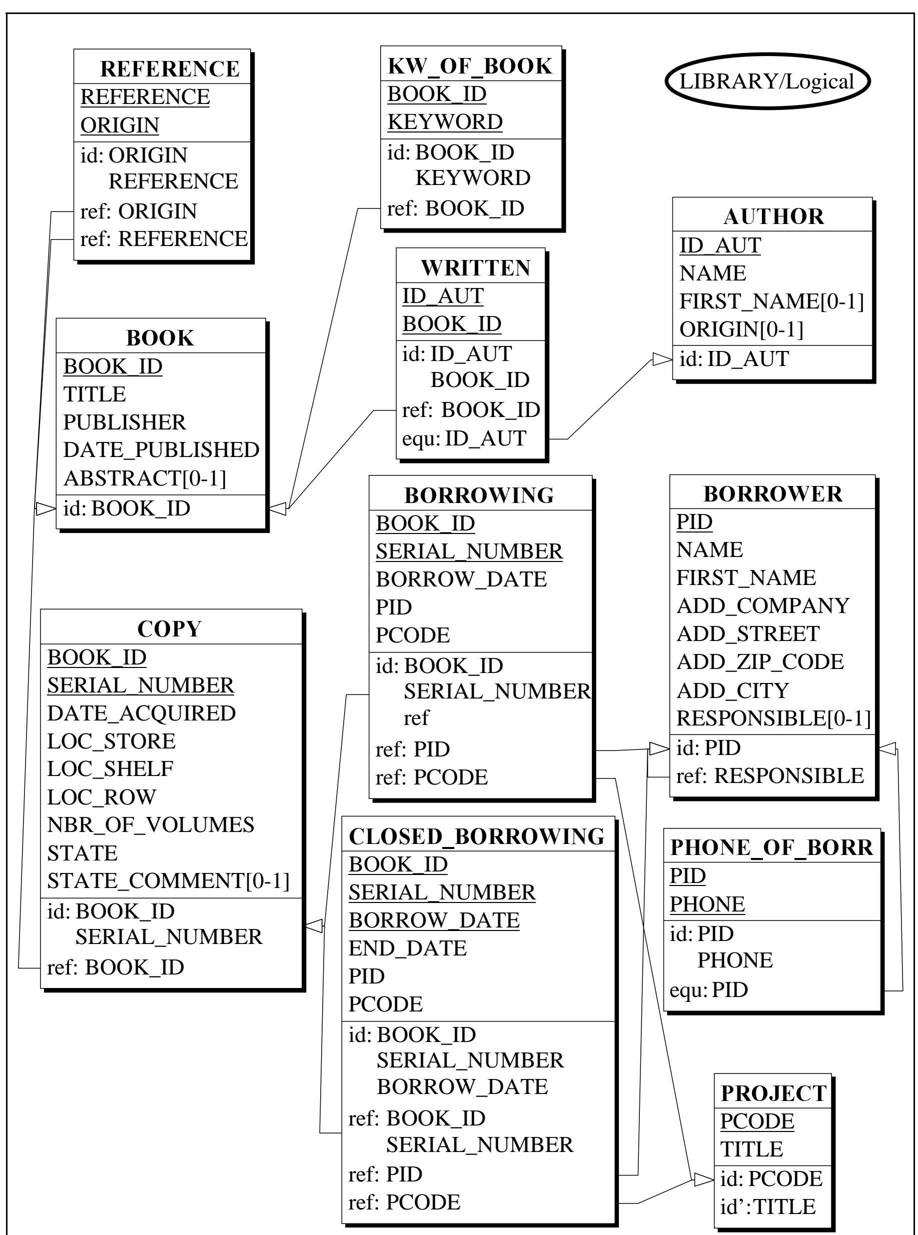


Figure 10.11 - The final relational logical schema, now fully SQL-compliant.

10.6 The names

The names of the entity types and of the columns directly derive from conceptual names. They do not necessarily satisfy the naming conventions of the DBMS. For instance, some characters may be invalid, or reserved words must be avoided.

The current logical schema includes an invalid symbol, namely "-" (dash), which is prohibited in most SQL DBMS. The most elegant solution is to replace all its occurrences by the symbol "_" (underscore). Though it is not the case in our schema, such names as `TABLE`, `INDEX`, `DATE` should be replaced with any other words which do not appear in the reserved word list of the DBMS.

To complete the process, we must change the names as required by the SQL syntax. We proceed as suggested in Section 5.9 (Figure 10.11).

10.7 Quitting the lesson

We save the current project under the name `logical-10.lun` and we quit DB-MAIN.

Technical addenda

10.8 On the equivalence of *Instance* and *Value* representations

We have considered two distinct techniques for transforming a multivalued attribute. Though other techniques exist, these ones are the most important. Therefore, it is essential that we analyze them in order to get a better understanding of their properties. The array of Figure 10.12 summarizes all the variants of multivalued attribute transformation into entity type, so that the equivalence of *Instance* and *Value* representations can be perceived more clearly.

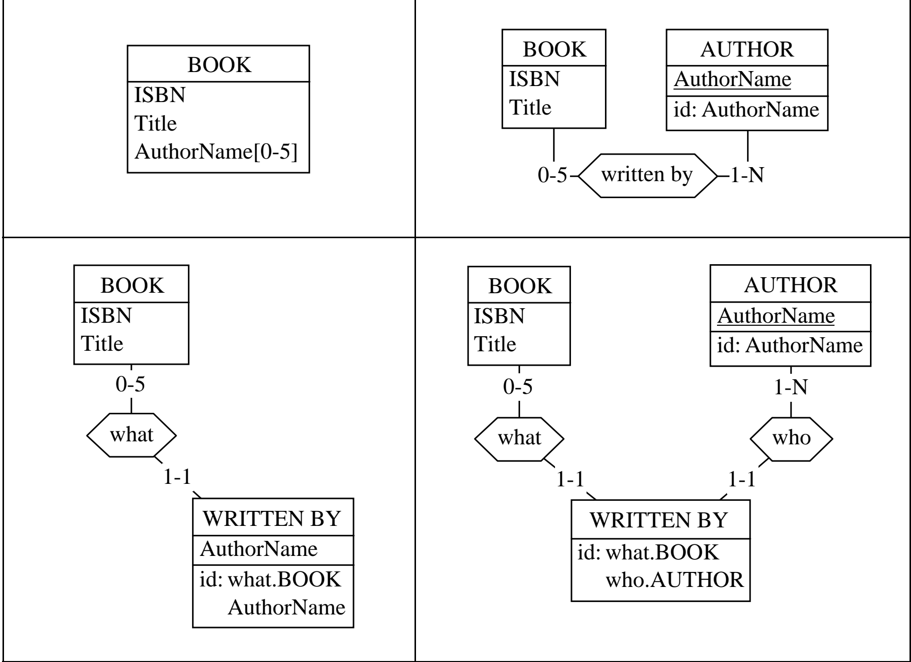


Figure 10.12 - The equivalence array of four versions of a multivalued attribute. It has been obtained by applying the two version of *attribute/entity type* transformation and the *rel-type/entity type* transformation.

10.9 On transforming *compound* attributes

The processing of compound attributes `Location` and `Address` was based on the *disaggregation* transformation. This is a simple and intuitive technique. However, it has a major drawback, it hides the initial grouping of the components, despite the pathetic use of prefix which suggests the lost aggregate.

In fact, other techniques exist. Two of them consists in transforming the compound attribute into an entity type, either by *Instance* representation, or by *Value* representation. Let us try both techniques on `Location` of `COPY`, in the abbreged version of Figure 10.13 (left).

First, `Location` is extracted by representation of its instances: each instance of `Location` (i.e., one for each `COPY` instance) is represented by a `LOCATION` entity. The rel-type is *one-to-one*, and makes `COPY` the implicit identifier of `LOCATION` (Figure 10.13).

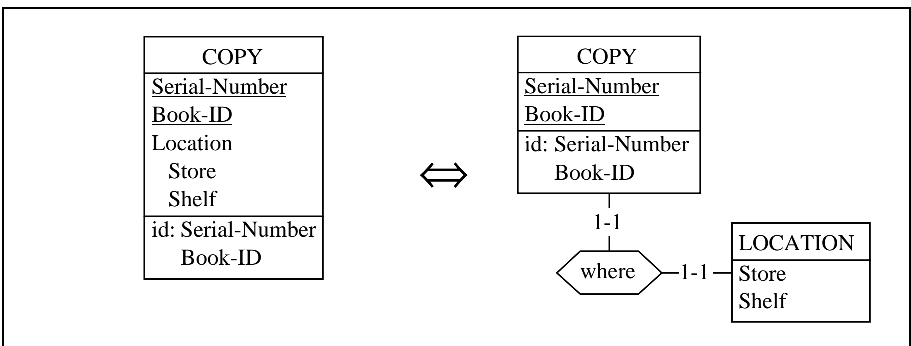


Figure 10.13 - Extracting a compound attribute as an entity type (*Instance* representation).

Then, this rel-type is reduced to a foreign key, which in addition becomes the explicit identifier of `LOCATION` (Figure 10.14).

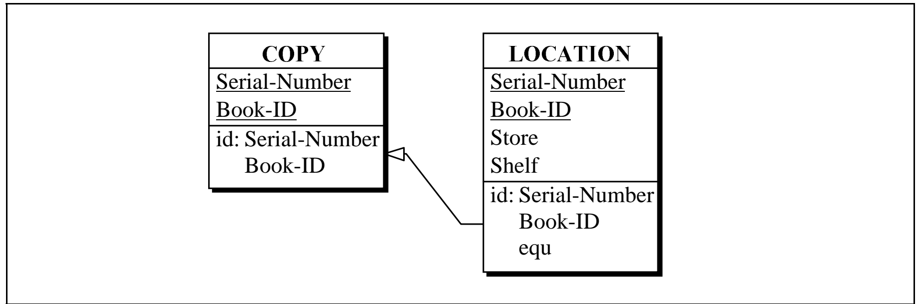


Figure 10.14 - Another relational expression of a compound attribute.

Let us try the *Value representation* technique on `Location` (Figure 10.15). Now, there is one `LOCATION` entity for each **distinct** value of `Location`, wherever it appears in `COPY` entities. There can be several `COPY` entities for one `LOCATION` entity. In other words, the entity type `LOCATION` is a dictionary of book locations, hence its identifier: (`Store`, `Shelf`).

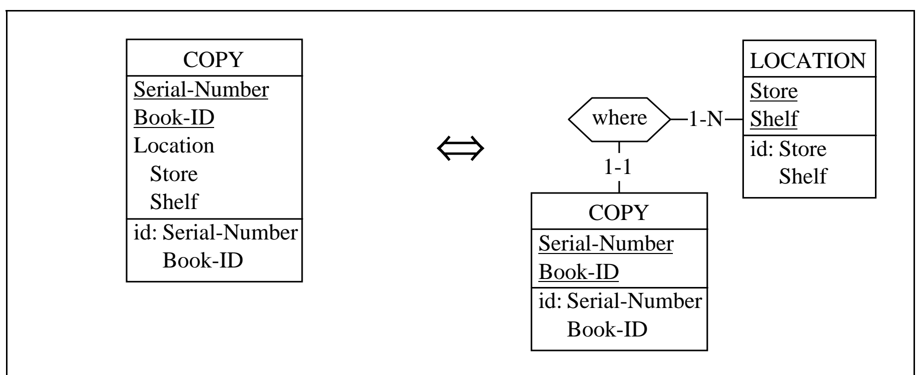


Figure 10.15 - Extracting a compound attribute as an entity type (*Value* representation).

However, transforming the rel-type `where` into a foreign key is not so obvious. Do try to transform it: DB-MAIN includes into `COPY` a foreign key made of the identifier of `LOCATION`, i.e., all its attributes (Figure 10.16). Not a particularly elegant and concise result, when compared with the source schema!

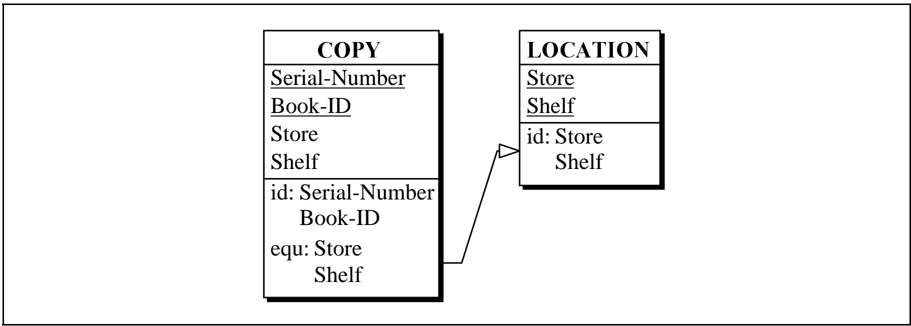


Figure 10.16 - This relational schema appears useless.

We must proceed differently. The identifier of `LOCATION` is long and complex. The situation would be better if this entity type had a short and simple identifier. Let us give it such an identifier through the command **Transform** / **Entity type** / **Add Tech ID** (Figure 10.17).

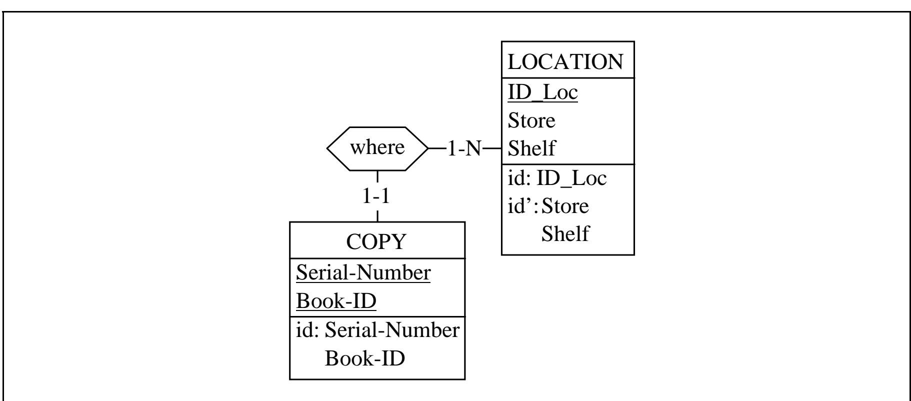


Figure 10.17 - Adding a technical ID to `LOCATION`.

Now, transforming the rel-type into a foreign key is straighforward and gives a more elegant result (Figure 10.18). One problem is that the new technical attribute must be correctly managed through a value constructor which delivers a new value for `ID_Loc` each time it is called. Most relational DBMS offers data types with that property[3].

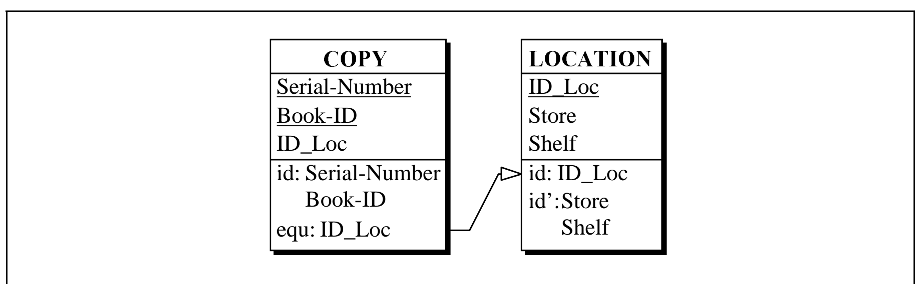


Figure 10.18 - A third relational expression of a compound attribute.

3. Such as the SEQUENCE feature of Oracle.

Summary of Lesson 10

- In this lesson, we have discussed
  - the various ways to process a *multivalued* attribute, and we have proved the equivalence of these techniques
  - the various ways to process a *compound* attribute, and we have proved the equivalence of these techniques.

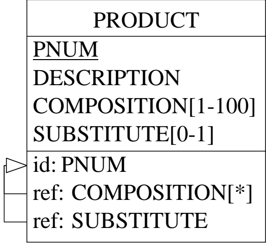
Exercises for Lesson 10

10.1 Propose a relational logical version for the following entity type.

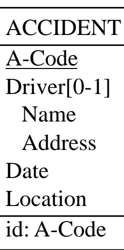
```
COMPANY
Com-ID
Com-Name
Com-Address
  Number[0-1]
  Street
  City
    Postal-Code[0-1]
    City-Name
Com-Revenue[0-1]
Phone[1-4]
  Country[0-1]
  Area
  Local
```

10.2 Modify the result of question 10.1 in such a way that the schema does not include any optional columns. *Hint*: use the *attribute/entity type* transformation to get rid of optional attributes.

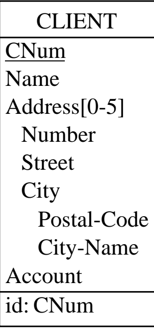
10.3 Find a conceptual expression for the following schema (this is a reverse engineering exercise).



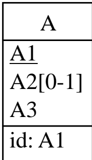
10.4 Transform the following schema into relational structures.



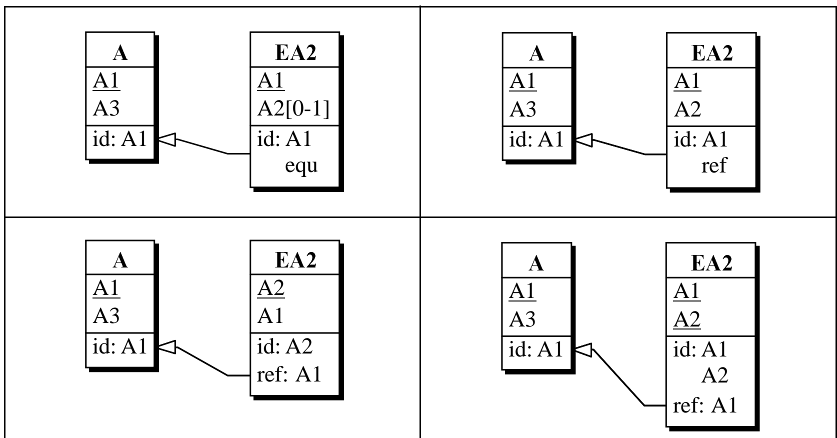
10.5 Transform the following schema into relational structures.



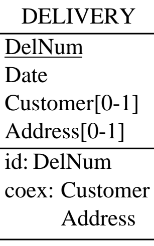
10.6 Consider the following schema.



Which of the SQL-compliant schemas proposed below can be conside-red strictly equivalent to the former?

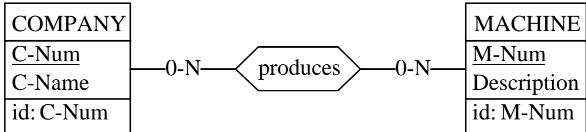


10.7 Propose a SQL-compliant schema in which the following coexistence constraint has been transformed

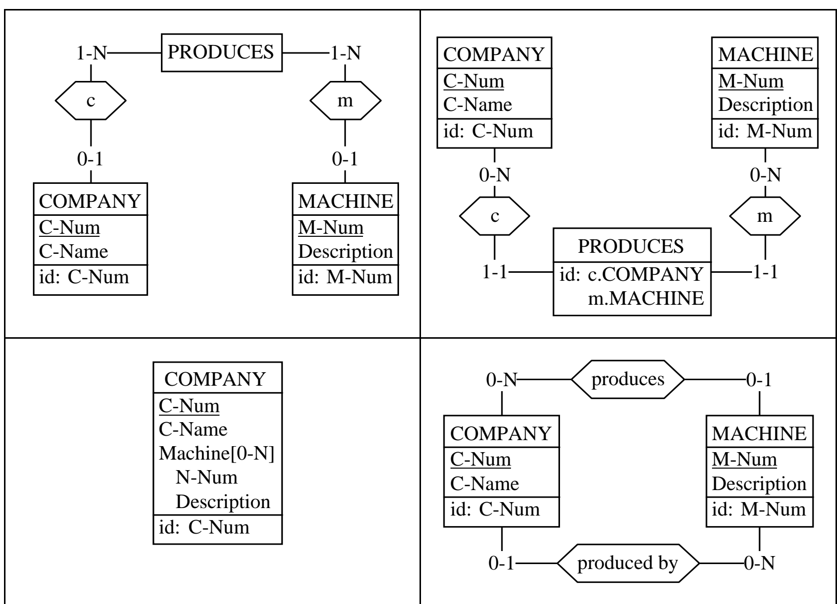


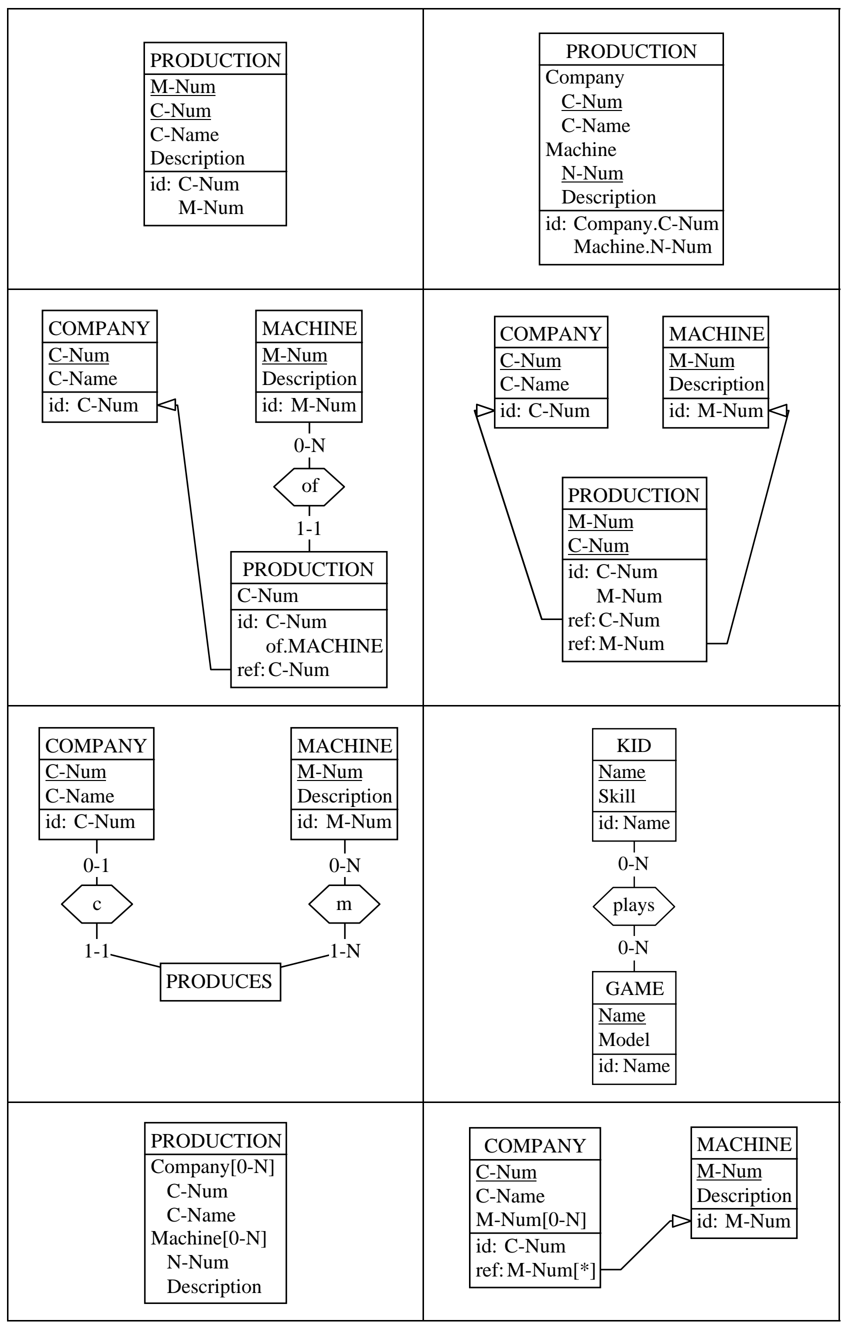
10.8 Prove that the following entity types are (*or* are not) equivalent.

10.9 Consider the following conceptual schema.



A database designer thinks he has found twelve equivalent schemas. Can you help him in choosing those which are really equivalent to the former? Prove your choice.




PRODUCTION
M-Num
C-Num
C-Name
Description
id: C-Num
M-Num
PRODUCTION
Company
C-Num
C-Name
Machine
N-Num
Description
id: Company.C-Num
Machine.N-Num
COMPANY
C-Num
C-Name
id: C-Num
MACHINE
M-Num
Description
id: M-Num
0-N
of
1-1
PRODUCTION
C-Num
id: C-Num
of.MACHINE
ref: C-Num
COMPANY
C-Num
C-Name
id: C-Num
MACHINE
M-Num
Description
id: M-Num
PRODUCTION
M-Num
C-Num
id: C-Num
M-Num
ref: C-Num
ref: M-Num
COMPANY
C-Num
C-Name
id: C-Num
MACHINE
M-Num
Description
id: M-Num
0-1
c
1-1
0-N
m
1-N
PRODUCES
KID
Name
Skill
id: Name
0-N
plays
0-N
GAME
Name
Model
id: Name
PRODUCTION
Company[0-N]
C-Num
C-Name
Machine[0-N]
N-Num
Description
COMPANY
C-Num
C-Name
M-Num[0-N]
id: C-Num
ref: M-Num[*]
MACHINE
M-Num
Description
id: M-Num

Lesson 11

Logical Design (3)

Objective

This lesson describes a systematic transformation plan which garantees the production of fully SQL-compliant schemas from most conceptual schemas. It completes and organizes the operations described in Lessons 9 and 10. The `LIBRARY` conceptual schema is processed according to this plan.
This lesson also describes three assistants that can help analyze and transform schemas. In particular, they provide developers with an easy way to write reusable scripts that automate the analysis and the production of schemas.

11.1 Starting Lesson 11

We start DB-MAIN and we open the project `logical-10` which now includes the conceptual schema and the logical schema of the database in project. We save it as `logical-11.lun`, the version on which we will work in this lesson, then we delete the schema `LIBRARY/Logical`, so that only the conceptual schema remains

We build a new schema, called `LIBRARY/Logical` (once again), by copying the conceptual schema. We open this (future) logical schema.

11.2 Working more systematically

In the previous two lessons, we have produced a SQL-compliant schema by applying a series of transformations. In this lesson, we will revisit in deeper detail the way in which we obtained this schema, we will improve this procedure, and finally automate it.

We can ask two questions about this procedure:

- was it the best way to proceed?
- does this procedure work with any conceptual schema?

Unfortunately, the answer to both questions is NO. Let us examine some problems which escaped our attention:

- reducing *one-to-many* rel-types into foreign keys occurs in several places; wasn't it possible to do it only once?
- processing a multivalued attribute may push a compound attribute at level 1[1];
- conversely, processing a compound attribute may push a multivalued attribute at level 1; the schema of Figure 11.1 illustrates the last two points: transforming `Address` makes `City` and `Phone` level 1 attributes.

1. We call *level 1* the level of the attributes directly attached to the entity type (or the rel-type). In a SQL-compliant schema, all attributes are at the level 1. The direct components of a level 1 attribute are at level 2, and so on.

 DB-MAIN Mini-tutorial - © J-L Hainaut 2002

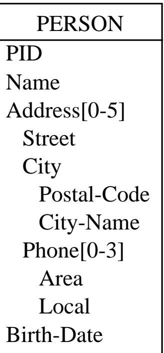


Figure 11.1 - Multi-level attributes.

- transforming a *one-to-many* rel-type may be impossible due to a still unresolved identifier, i.e., an identifier that includes a role of a rel-type which has not been transformed yet. In the example of Figure 11.2, `works-in` cannot be processed until the rel-type `from` has been transformed into a foreign key, making the identifier of `BRANCH` all-attribute.

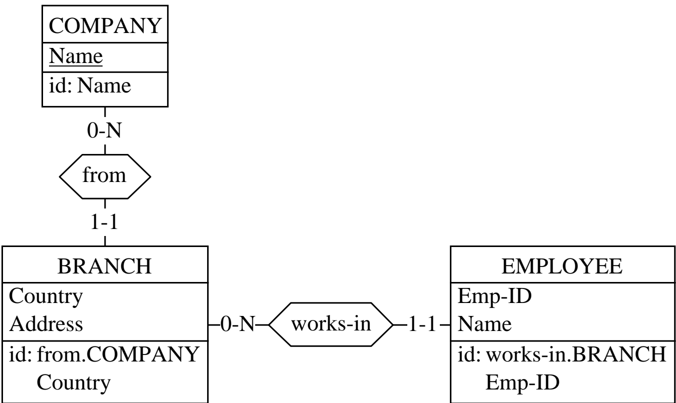


Figure 11.2 - Rel-type `works-in` cannot be transformed before `from`.

- what about IS-A relations? Since they are not SQL-compliant, they must be transformed into equivalent relational structures.

Obviously, we need to complete our initial intuitive procedure, and to organize its steps in a more systematic way. Since processing IS-A relations is a quite new problem, we will tackle it first.

11.3 Transforming the IS-A relations

Supertype/subtype hierarchies (through IS-A relations) are very powerful semantic constructs, but they have no direct representation in standard DMS, such as SQL[2]. In addition, mastering them poses some interesting, but rather complex problems that would lead us far beyond the objectives of this introductory volume.

There are several ways to replace IS-A relations into equivalent plain data structures. Each of them comes with its drawbacks and its advantages, and none can be claimed to be superior to the others on all the possible criteria. We will present a limited, but intuitive technique which can be easily carried out. It applies to IS-A structures on which no constraints D (disjoint) and T (total) are defined, or, more precisely, whose constraints D and T will not be translated. In other words, the subtypes may overlap, and the supertype is partially covered by its subtypes. More complex situations require advanced techniques which will be ignored in this lesson (though a short discussion will be proposed in the Addenda.

The proposed technique consists in representing each concerned entity type, be it supertype or subtype, by an *independent entity type*, and in connecting each former subtype to its supertype by a *one-to-one* rel-type as illustrated in Figure 11.3.

2. At least in SQL2. The next standard (SQL3 or SQL-1999) includes the concept of type hierarchy, inducing a sort of IS-A relation on the tables of a database. We will ignore this feature in this volume.

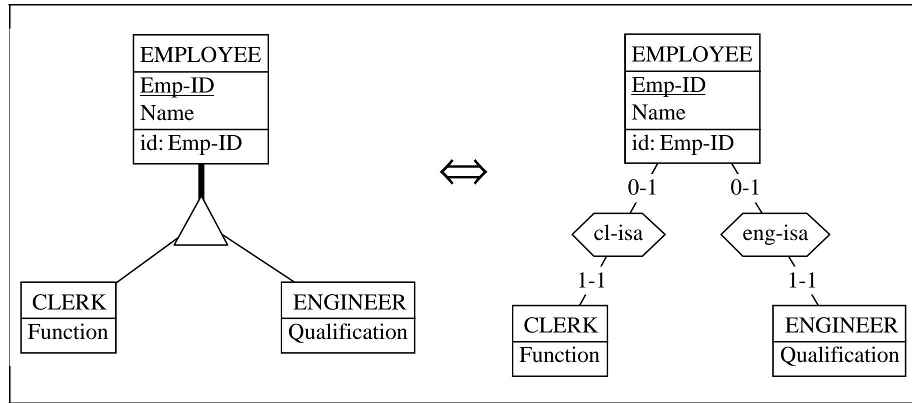


Figure 11.3 - Representation of IS-A hierarchy through *one-to-one* rel-types.

Translating this schema into pure SQL-compliant structures is now easy (Figure 11.4). It is important to note that each foreign key is an identifier as well. From the user point of view, the supertype entities are stored in the table `EMPLOYEE`, while the subtype entities must be rebuilt by joining the subtype table with the supertype table: data about *clerks* are obtained by the natural join between `CLERK` and `EMPLOYEE`.

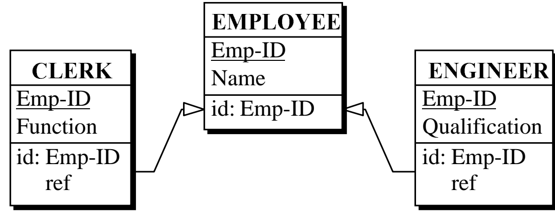


Figure 11.4 - SQL-compliant structure translation an IS-A hierarchy.

This implementation does not enforce any *disjunction* constraint between the contents of `CLERK` and `ENGINEER`, nor any *totality* constraint on `EMPLOYEE`. Indeed, a given employee entity can also appear in the table `CLERK`, in the table `ENGINEER`, in both or even in none.

11.4 A transformation plan

With these considerations in mind, we can propose a general procedure to translate conceptual schemas in SQL-compliant structures.

Transformation plan for relational schemas	
Step 1.	Transform the **IS-A relations** into one-to-one rel-types
Step 2.	Transform **complex** and **many-to-many rel-types** into one-to-many (or one-to-one) rel-types
Step 3.	Transform level-1 single-valued **compound attributes** by disaggregation
Step 4.	Transform level-1 **multivalued attributes** into entity types (instance representation)
Step 5.	Repeat steps 3 and 4 until there are no more compound or multivalued attributes
Step 6.	Transform **one-to-many** (and one-to-one) **rel-types** into reference attributes
Step 7.	Repeat step 6 until no more rel-types can be transformed
Step 8.	For every rel-type which could not be transformed in step 7 due to the absence of entity type identifier, **add a technical identifier** to the concerned entity type
Step 9.	Repeat steps 6 and 7 until no rel-types remain.
Step 10.	Translate the names according to the SQL syntax.

Figure 11.5 - A detailed procedure to translate conceptual schemas into SQL-compliant schemas.

In this plan, the transformation of *one-to-many* (and *one-to-one*) rel-types into reference attributes is delayed until all the transformations that may produced such rel-types (Steps 1, 2 and 4) have been carried out.

In addition, processing the compound and multivalued attributes is repeated to cope with nested attributes.

Finally, the blocking structures preventing the transformation of *one-to-many* (and *one-to-one*) rel-types are processed in two ways: by iterating on this transformation (hybrid identifier problem), and by adding technical identifiers when necessary (missing id problem).

Now let us apply this transformation plan to the conceptual schema we developed in Lesson 8.

Step 1: the IS-A relations

The schema has no IS-A structures.

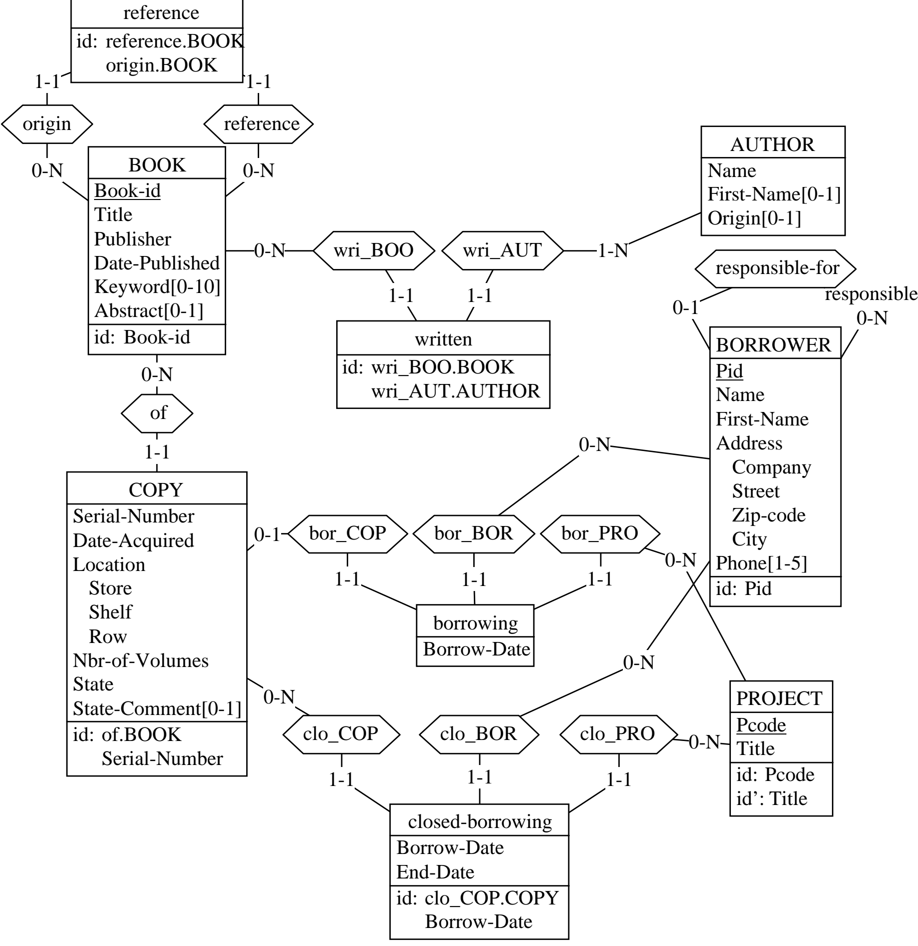


Figure 11.6 - Complex rel-types transformed.

Step 2: complex and many-to-many rel-types

All the rel-types with a degree (number of roles) greater than 2, or with attributes, are called complex. The complex rel-types `borrowing` and `closed-borrowing` are transformed into entity types.

We process the *many-to-many* rel-types in the same way. Now, `reference` and `written` are replaced with equivalent entity types. The result is shown in Figure 11.6.

Step 3: compound attributes

The compound attributes `Location` and `Address` are disaggregated, i.e., replaced by their respective components. Hence the schema of Figure 11.7.

Step 4: multivalued attributes

The multivalued attributes `Keyword` and `Phone` are transformed into entity types (Figure 11.8).

Step 5: repeating steps 3 and 4

There are no more compound or multivalued attributes. Therefore, this step does not apply.

Step 6: one-to-many rel-types

Let us suppose that we process the rel-types in the alphabetical order of their names: `bh`, `BOO_key`, `bor_BOR`, `bor_COP`, `bor_PRO`, `clo_BOR`, `clo_COP`, `clo_PRO`, `of`, `origin`, `reference`, `responsible-for`, `wri_AUT`, `wri_BOO`.

We observe that some rel-types have not been reduced. For instance, the transformation of BOR_COP and CLO_COP failed because the referenced entity type, `COPY`, still had a hybrid identifier comprising the role `of.BOOK`. Hence the following step.

Step 7: repeat step 6

In fact, the rel-type `of` have been reduced at the end of Step 6, but too late to allow BOR_COP and CLO_COP to be transformed. Now it does not block the transformation of these rel-types any longer. So we apply once again the transformation of step 6, which correctly transforms the rel-types BOR_COP and CLO_COP (Figure 11.10).

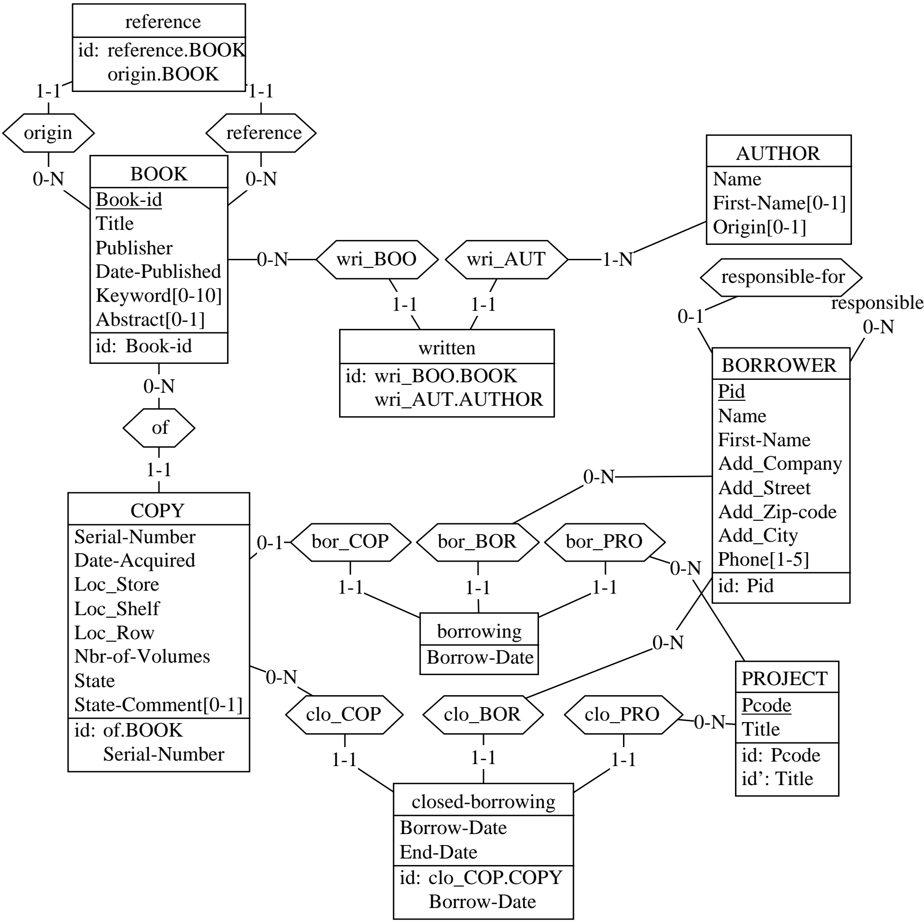


Figure 11.7 - Compound attributes disaggregated.

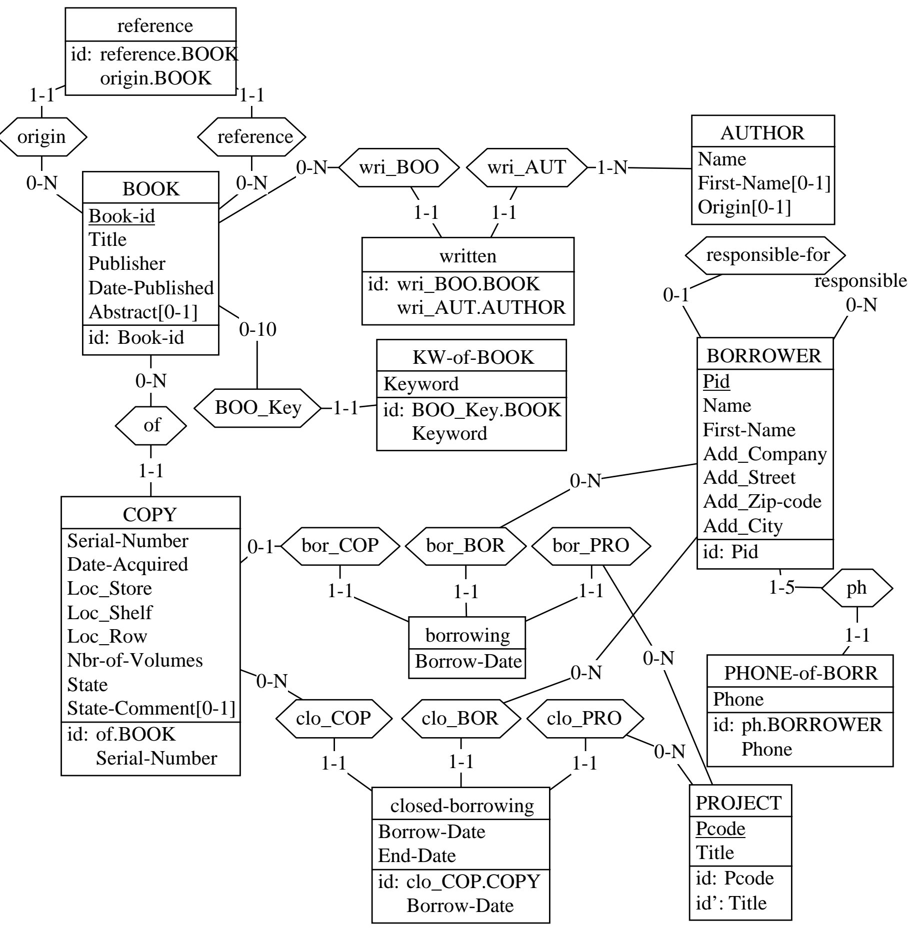


Figure 11.8 - Multivalued attributes extracted as entity types.

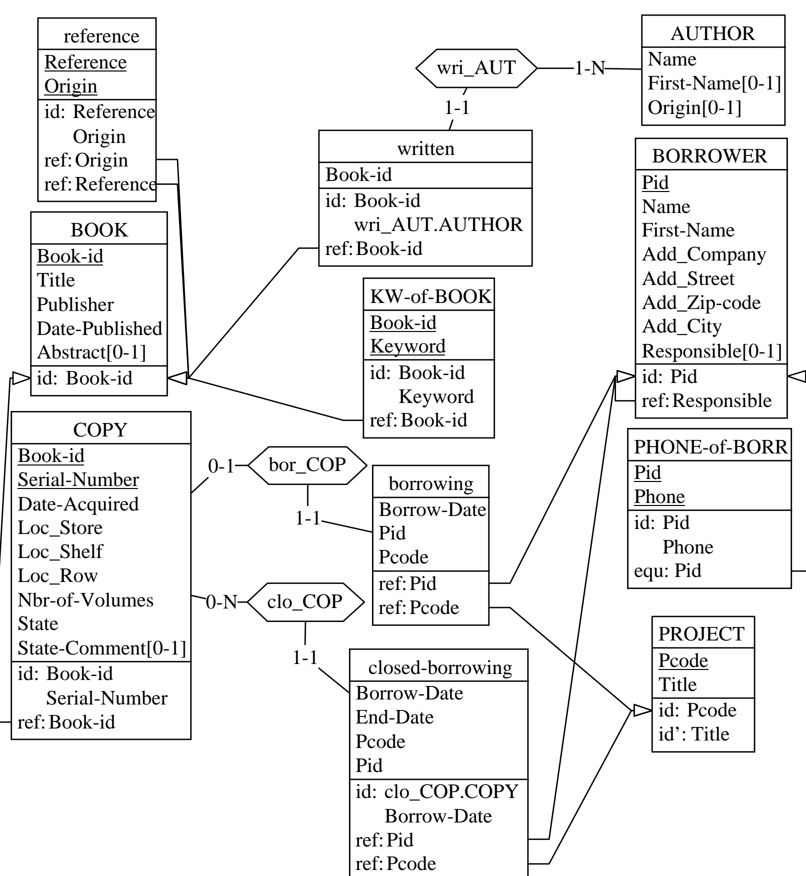


Figure 11.9 - Simple rel-types transformed into foreign keys. Three rel-types fail to be transformed.

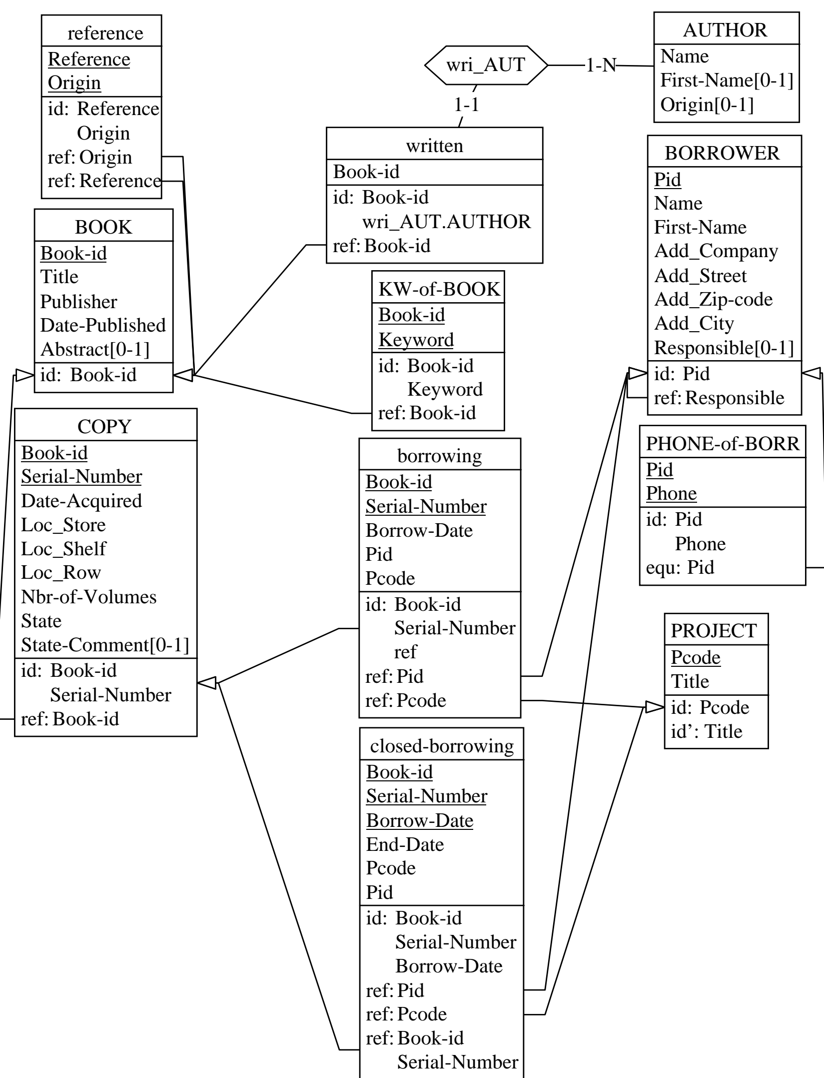


Figure 11.10 - Remaining simple rel-types transformed into foreign keys. One rel-type remains however.

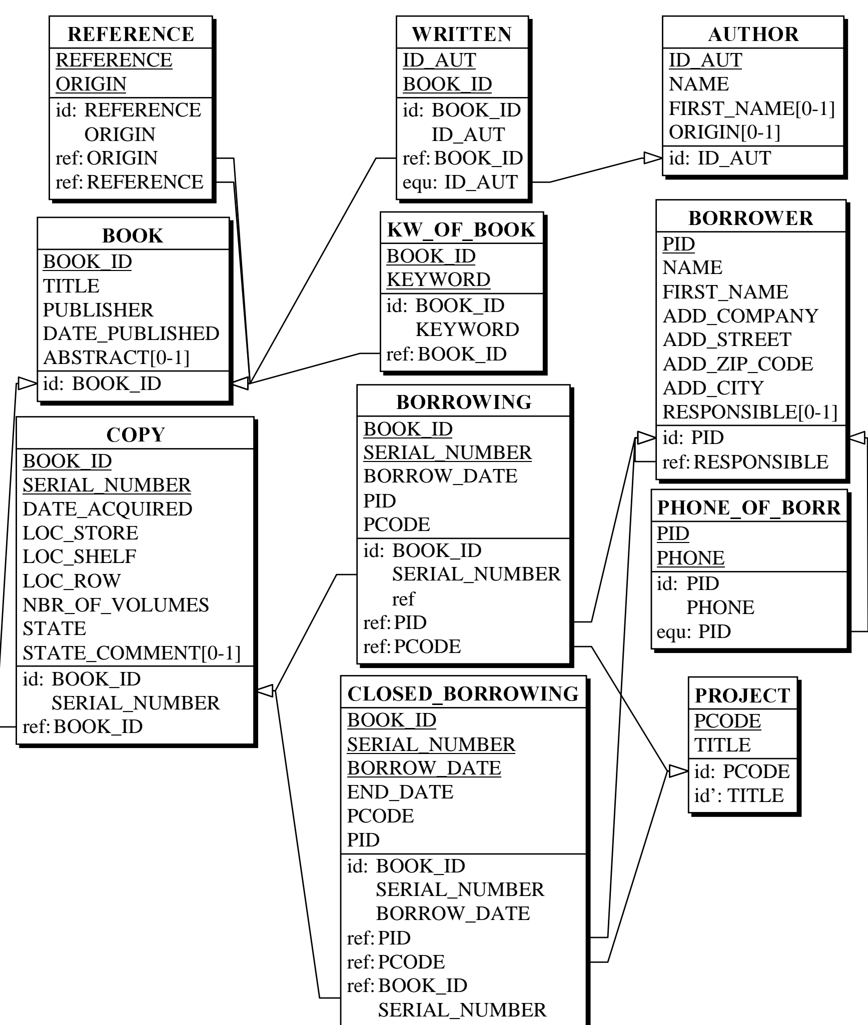


Figure 11.11 - The SQL-compliant logical schema.

WRI_AUT still remains, but for another reason. It is not blocked due to hybrid identifiers, but because of the lack of identifier in AUTHOR.

Step 8: missing identifiers

We add a technical identifier to AUTHOR (Figure 11.12).

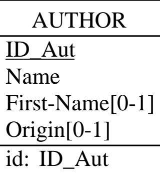


Figure 11.12 - Adding a technical ID to AUTHOR.

Step 9: repeat step 6

... and we try again to reduce WRI_AUT. The structures are now fully SQL-compliant.

Step 10: Translate the names

We transform the names: uppercase, replace spaces and dashes, replace reserved words, etc. (Figure 11.11).

11.5 The Global transformation Assistant

This transformation plan seems to work correctly, but the way we applied it is rather tedious when processing large and complex schemas. DB-MAIN offers a very powerful processor to help us apply transformation plans, and even to write, save and reuse such plans.

To practice this assistant, we can delete the current logical schema and ask for another copy of the conceptual schema that we open. We call the assistant by the command **Assist** / **Global transformation**.

The assistant is made of three main parts, the left-side area is the problem solver, while the right-side area comprises script management functions and the control panel (Figure 11.13). For the present time, we will use the problem solver and the control panel.

The **problem solver** is structured in two columns.

The first column proposes a collection of potential **problems**, classified by object type (Entity type, Rel-types, Is-a, Attributes, Groups, Miscellaneous, etc.). When we select an object type, a list of typical situations is proposed (the *problems*), from which we can select one.

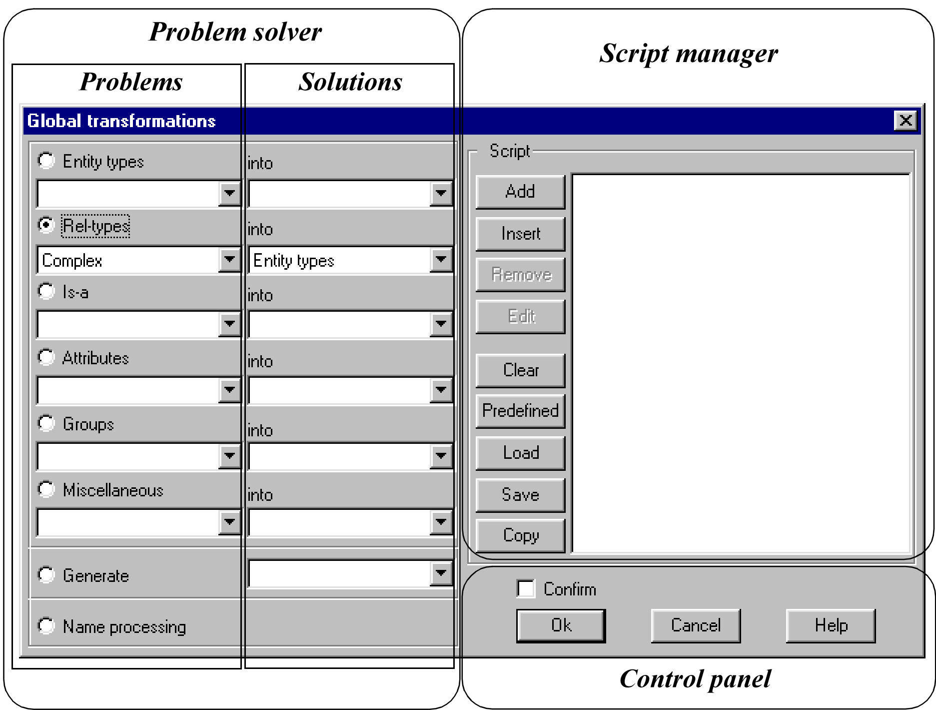


Figure 11.13 - The structure of the *Global transformation assistant*.

The second column, the **into** list, proposes a list of possible actions (mainly transformations) to solve the selected problem (the *solutions*), from which we select one.

By clicking on button OK in the control panel, we *execute the selected action on all the objects of the schema that correspond to the selected problem*. The confirm button allows us to control the process object by object.

However, the best way to learn how all this works is to use it for solving real problems. This is the objective of the next section.

The Problem solver

We will use this assistant to produce a SQL-compliant schema from the current conceptual schema. We will still proceed step by step, as suggested in the transformation plan of Figure 11.5.

Step 1: the IS-A relations

Should the schema comprised such structures, we could have used the assistant to help us (through the **Is-a** object type).

- We select the item **All** in the **Is-a** object type list (this is our *current problem*).
- We select the item **Rel-type** in the **Into** list (this is the *proposed solution*).
- We unchecked the Confirm button (automatic mode).
- We click on OK.

Step 2: complex and many-to-many rel-types

We proceed as follows (this operation is illustrated in Figure 11.13):

- We select the item **Complex** in the **Rel-type** object type list (our *current problem*).
- We select the item **Entity type** in the **Into** list (the *proposed solution*).
- We click on OK.

That's all: the complex rel-types `borrowing` and `closed-borrowing` have been transformed.

We now process the *many-to-many* rel-types in the same way:

- We select the item **Binary N-N** in the **Rel-type** object type list.
- We select the item **Entity type** in the **Into** list.
- We click on OK.

Now, `reference` and `written` are replaced with equivalent entity types. The resulting schema inludes *one-to-many* and *one-to-one* rel-types only.

Step 3: compound attributes

There is an operation for them:

- We select the item **Compound** in the **Attribute** object type list.
- We select the item **Disaggregation** (or **Entity type**) in the **Into** list.
- We click on OK.

`Location` and `Address` are replaced with their components.

Step 4: multivalued attributes

The multivalued attributes `Keyword` and `Phone` are transformed into entity types:

- We select the item **Multivalued** in the **Attribute** object type list.
- We select the item **Entity type** in the **Into** list.
- We click on OK.

Step 5: repeating steps 3 and 4

This step does not apply here.

Step 6: one-to-many rel-types

The *one-to-many* and *one-to-one* rel-types are transformed into entity types:

- We select the item **Binary 1-N** in the **Rel-type** object type list (it includes *one-to-one* rel-types as well).
- We select the item **Referential attributes** in the **Into** list.
- We click on OK.

Wherever possible, the *one-to-many* and *one-to-one* rel-types are transformed into reference attributes. This operator automatically repeats this transformation until no rel-types can be transformed in this way anymore.

Step 7: repeat step 6

This has been done by the problem solver.

Step 8: missing identifiers

We add a *technical identifier* to all the entity types which need one in order to make the transformation of rel-types into reference attributes possible (here AUTHOR).

- We select the item **Where needed** [3] in the **Entity type** object type list.
- We select the item **Add technical id** in the **Into** list.
- We click on OK.

Step 9: repeat step 6

That is:

- We select the item **Binary 1-N** in the **Rel-type** object type list.
- We select the item **Referential attributes** in the **Into** list.
- We click on OK.

---

3. This label tells that technical ids will be associated only with entity types that will become referenced tables.

This exercise is a good illustration of the concept of problem solving in DB-MAIN.

A **problem** is a family of structures which is perceived as a problem in the current context. For instance, a compound attribute is not a problem in itself, but it definitely is one when we try to build a SQL-compliant schema. In short, a problem is a construct considered *invalid* in a given context.

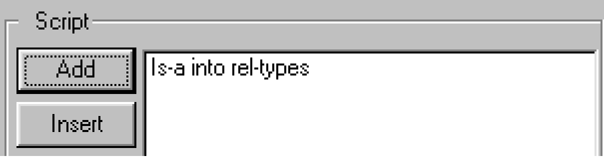
A **solution** to a problem is an action (generally a transformation) which operates on the *invalid* construct, and which replaces it by another construct. In general, there can be more than one solution. It is up to the analyst to choose the action which best fits his/her objectives.

In the next lesson, we will use other functions of the Problem solver.

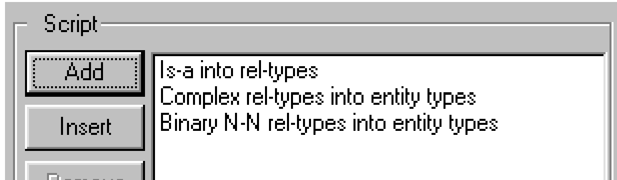
The Script manager

The best way to understand what a **script** is, consists in writing one by ourself. More precisely, we will automate the transformation plan used so far by writing the successive operations specified in the plan. Each operation can be specified in the *Problem solver* of the assistant by a problem/solution statement. However, these statements are not executed (as when we clicked on button OK), but they are stored in the *script area* of the assistant.

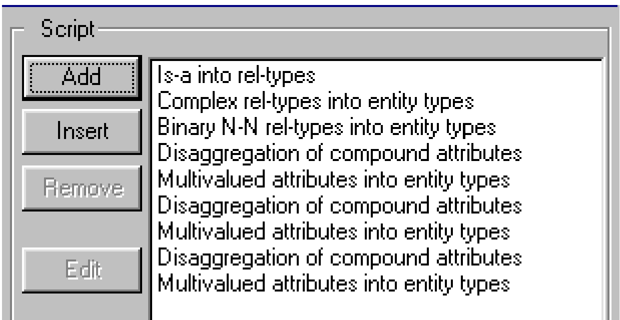
Let us consider the first step: *translating **IS-A** relations into one-to-one rel-types*. In the same way as we did in the previous section, we open the **Is-a** list in the Problem solver, and we select the **All** item (not really difficult since this is a single-item list!). Then we open the **Into** list, and we select the **Rel-types** item. We add this statement to the (currently empty) script area by clicking on the Add button. The script area now looks like the screen below.



To specify the second step, we select **Complex** in the **Rel-type** list, and the **Entity type** in the **Into** list, and we add this statement in the script area. We address the *many-to-many rel-types* in the same way. The script now includes three statements:



The steps related to attributes require a more subtile approach. Indeed, the scripting language does not offer loop structures, nor any standard control structures that are common in most programming languages[4]. So, to process complex attribute structures, we have to add as many statements as there can exist attribute levels. We suppose that three levels of nesting is realistic, therefore, we add three blocks of attribute processing statements as follows:



The other transformation are introduced easily (Figure 11.14).

All that remains to be done is to transform the names. For that, we click on the button Name processing in the *Problems* part, and we click on Add. Then, the *Name processing* panel opens (Figure 5.17 and Figure 5.18), so that we can set the parameters. Later on, we can change these parameters by selecting the last statement in the script area and by clicking on `Edit`.

This completes the script (Figure 11.15) which can be saved (button Save) for further reuse (button Load).

4. As we will see in the Technical addenda, the *Advanced global transformation* Assistant will give us an elegant way to write more sophisticated scripts.

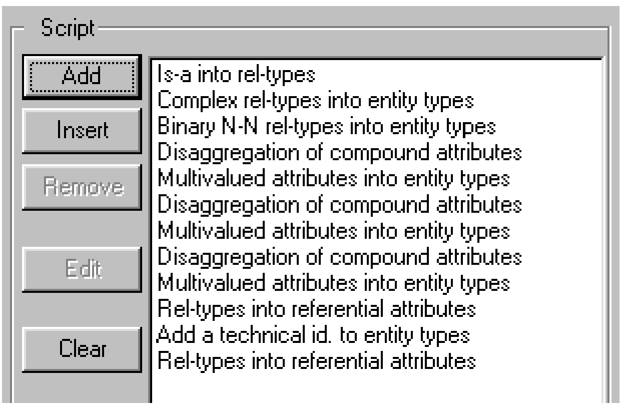


Figure 11.14 - Completing the transformations.

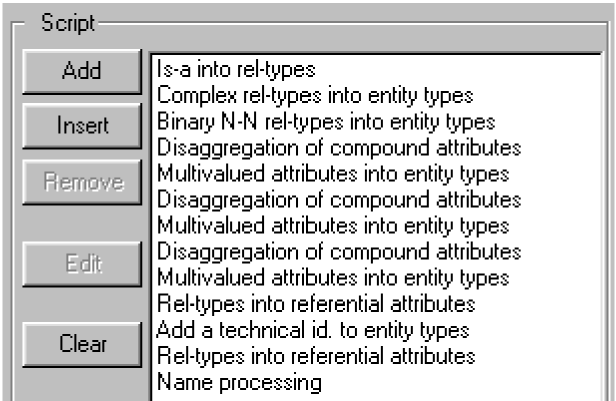


Figure 11.15 - The *Conceptual/Relational* transformation script.

To execute this script, we click on the button OK. Note that the effect of clicking on this button depends on the contents of the script area:

- if the *script area is empty*, the assistant executes the current problem/solution statement of the Problem solver,
- if the *script area contains a script*, the assistant executes this script instead.

The script manager offers the following script manipulations:

- button Add — adds the current problem/solution statement to the end of the current script
- button Insert — inserts the current problem/solution statement before the selected statement of the script (click on the statement to select it)
- button Remove — removes the current problem/solution statement
- button Edit — allows the modification of the parameters of the selected statement (e.g., Name processing)
- button Clear — empties the script area
- button Predefined — the assistant proposes predefined scripts to produce schemas for some standard models; clicking on this button includes the selected script in the script area; a good way to examine other examples of scripts;
- button Load — loads in the script area a script that has been saved on disk;
- button Save — saves on disk the contents of the script area.
- button Copy — saves the text of the script on the clipboard.

11.6 Quitting the lesson

We save the current project under the name `logical-11`. We can now quit DB-MAIN.

Technical addenda

11.7 IS-A transformation revisited

About the subtype constraints

In this lesson, we have proposed a general technique to get rid of IS-A relations without loss of information. As we carefully mentioned it, we ignored the subtype constraints D and T. What would have happened if we had processed IS-A complemented with such constraints?

Lesson 6 identified four patterns, illustrated in Figure 6.9, among which we can implement the last one only (¬**D** and ¬**T**). Let us experiment the translation of the **D** constraint (Figure 11.16). We observe that this constraint has been expressed as an `exclusive` constraint holding between the *one-to-one* rel-types. Indeed, it states that any `CUSTOMER` entity can be linked with one `PERSON` entity or with one `COMPANY` entity, but not both.

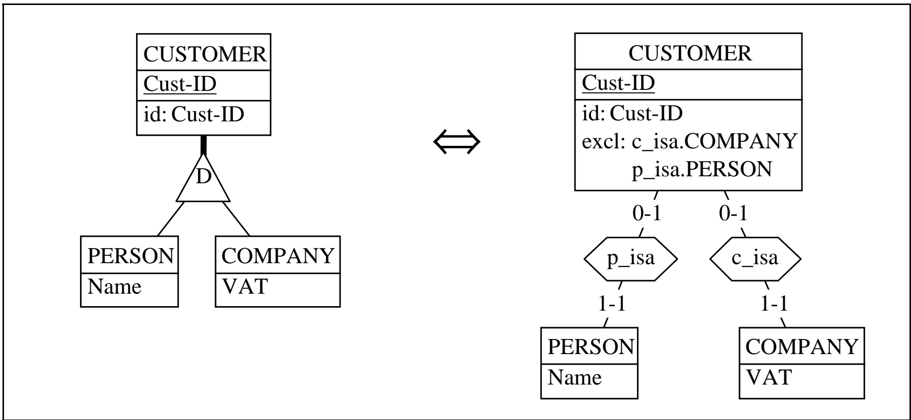


Figure 11.16 - Expression of an IS-A hierarchy with a D constraint.

Similarly, a **T** constraint would have been translated into an `at-least-one` constraint and a **P** constraint into an `exactly-one` constraint.

These translation rules preserve the subtype constraints. In particular, you can try the inverse transformation by yourself: select `CUSTOMER`, then execute **Transform / Entity type / Rel-types -> is-a**.

However, things get disappointing when we try to translate these rel-types into foreign keys. Indeed, the new constraints disappear, since the standard SQL-DDL does not offer any declarative clause to enforce them.

The DB-MAIN transformation tries to compensate this loss by introducing a new optional attribute in the supertype for each subtype, and applying the exclusive constraint to these attributes (Figure 11.17). The idea is that when a customer is a person, the `PERSON` attribute of its entity is set to a non-null value, while this attribute is void if the customer is not a person. This technique is better than nothing, but the correct management of these type attributes relies entirely on the user/programmer[5].

Other subtype constraints are translated similarly.

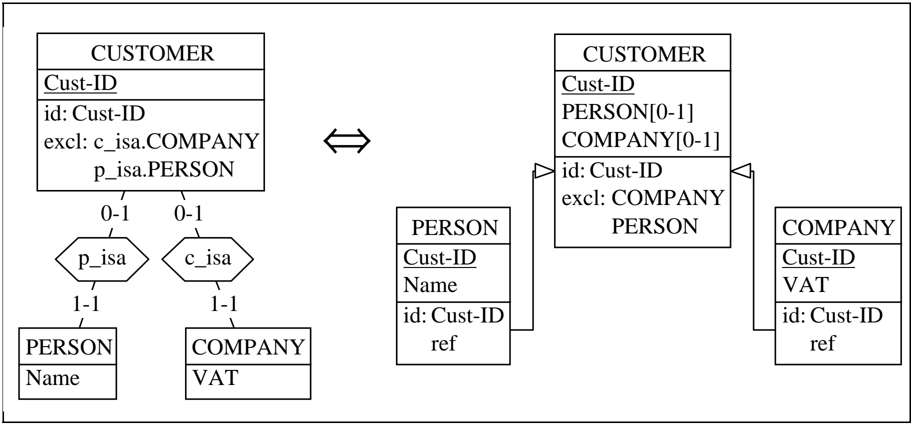


Figure 11.17 - An awkward artifact to simulate the *exclusive* constraint.

The *upward inheritance* technique

Several other techniques exist to translate IS-A hierarchies in standard structures. One of them can be derived from the basic technique we developed above, provided each subtype has a rather simple structure.

5. They can be managed by procedural fragments in *Triggers* for instance. However, writing correct triggers for such patterns requires special care.

According to this second technique, called *upward inheritance*, each subtype is integrated into its supertype, generally by transformation into an attribute. Considering the schema of Figure 11.16 (right), we apply the transformation **Transform / Entity type / -> Attribute** to each subtype. The result is presented in Figure 11.18 as the transformation of an IS-A hierarchy. It enjoys two qualities that the first technique lacks:

- the relational structure includes a correct translation of the **D** constraint,
- this constraint is easily encoded into a row-level `check` predicate.

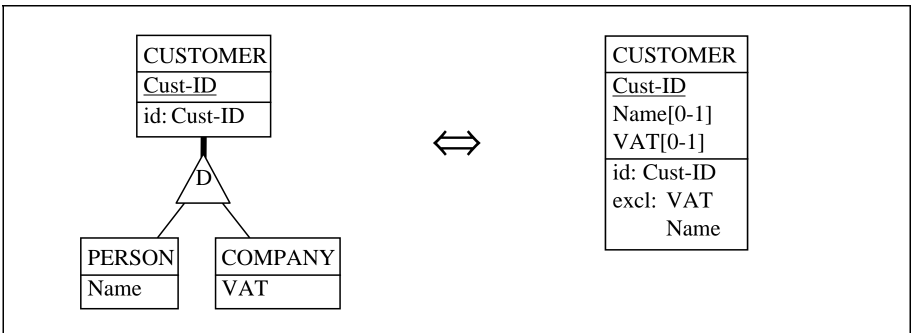


Figure 11.18 - Integrating the subtypes into the supertype. The subtype constraints are correctly translated.

Though further comparing these techniques would prove too technical for the scope of this tutorial, we will draw attention on the complexity of the translation of richer subtypes.

Let us consider that each subtype includes several attributes. The transformation of the subtypes into supertype attributes leads to *optional compound* attributes (Figure 11.19). Disaggregating these attributes to make them SQL-compliant yields complex integrity constraints (Figure 11.20).

Of course, it is possible to simplify the schema by observing that the `excl` constraint need not hold among the two group of attributes, because expressing this constraint among one attribute per group is quite equivalent[6] (Figure 11.21).

Nevertheless, the final schema, though correct and not too complicated to code in SQL, remains fairly complex, even for simple conceptual schemas.

6. Can you prove that this simplification is valid?

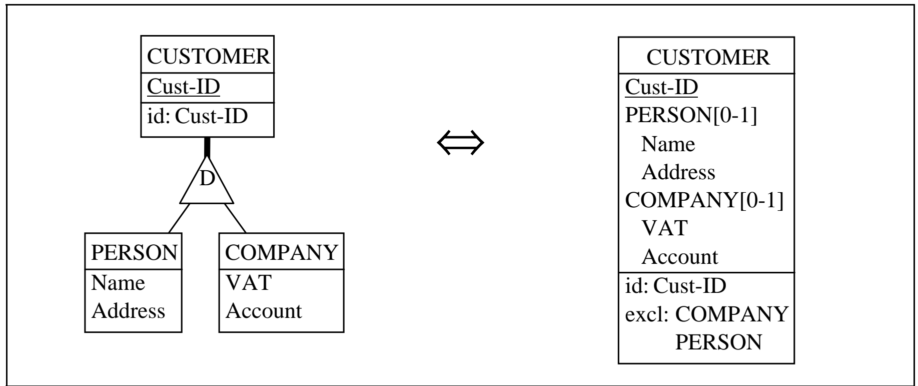


Figure 11.19 - Transforming an IS-A hierarchy with more complex subtypes.

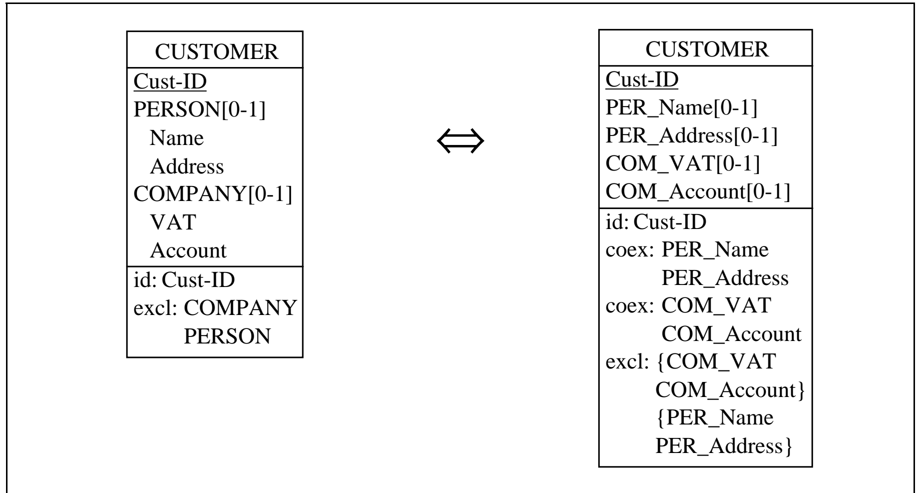


Figure 11.20 - Disaggregating the compound attributes leads to a complex IS-A hierarchy translation.

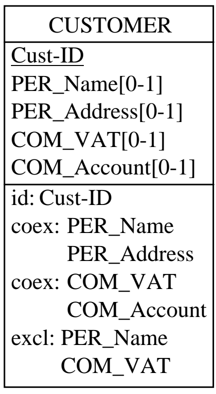


Figure 11.21 - Simplifying the `excl` constraint.

The *downward inheritance* technique

This third technique consists in copying the attributes (as well as roles and constraints) of the supertype into each of its subtype, then in discarding this supertype. In Figure 11.22 we process a slightly different source schema to better illustrate the technique.

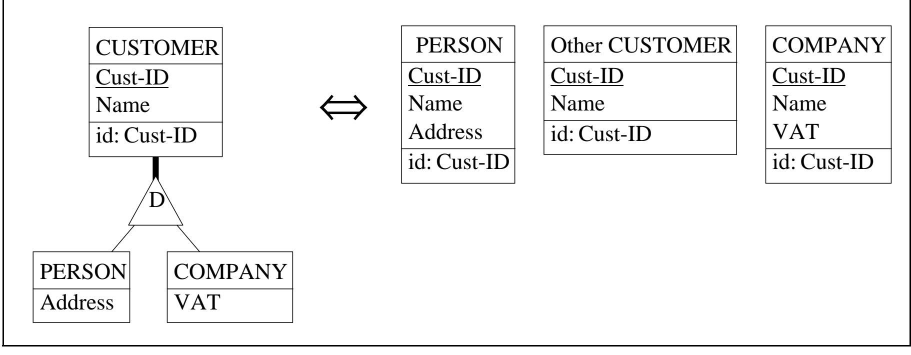


Figure 11.22 - Representing the subtypes only + the customers that fall in no subtypes (`Other CUSTOMER`).

In the right-side schema, we keep the subtypes `PERSON` and `COMPANY`, enriched with the attributes, the roles and the constraints of the supertype. Howe-

ver, since the hierarchy is not **T**otal, some entities may be of type CUSTOMER without being of any subtype. Hence the complementary entity type Other CUSTOMER which collects these entities. Quite naturally, Cust-ID is the primary identifier of each resulting entity type.

This technique is nice when the users are mainly interested in querying subtype entities rather than the supertype entities.

Unfortunately, the problem is a bit more complex. Indeed, the constraints that hold in the population of the supertype may not be preserved when distributed among the subtypes. This is the case for the primary id Cust-ID. Indeed, merely stating that this attribute is the primary id of each subtype is not sufficient. It must be an identifier of *the union of their population* as well. For instance, we cannot create a PERSON entity with a value of Cust-ID that already exists in COMPANY or in Other CUSTOMER. This constraint is not easy to define. In Figure 11.23, we express it as a kind of *non-inclusion* constraint, which states that each value of Cust-ID of PERSON is **not in** the set of values of Cust-ID of Other CUSTOMER, nor in that of COMPANY[7]. In this way, we can guarantee that the sets of Cust-ID from all the entity types are disjoint.

This constraint can be implemented as *triggers*[8] associated with each table resulting from these subtypes.

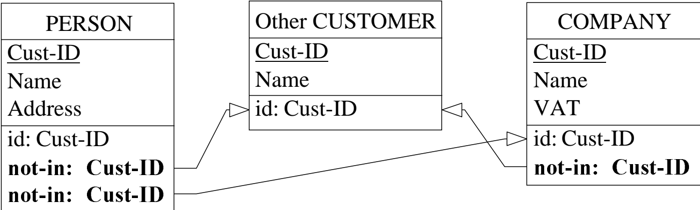


Figure 11.23 - The correct translation of Cust-ID being a primary identifier of the supertype CUSTOMER.

7. The constraint **not-in** is not built-in in the DB-MAIN tool. Instead, it has been defined as a generic constraint. More of this in the Tutorial *Computer-assisted Database Engineering - Volume 1: Database Models*.
8. Note that each trigger must check the *not-in* constraint with the N-1 other tables, where N is the number of subtypes.

If the *subtypes are not disjoint*, the situation is even more complex, since the same value of Cust-ID may appear in more than one subtype. In such a case, the associated values of Name (and of all the other supertype attributes and roles) **must be the same**. This pattern is illustrated in Figure 11.24. The subtypes have been declared **T**otal (to get rid of the Other CUSTOMER entity type) and not **D**isjoint. The additional constraint states that, when considering the **union** of the population of PERSON and COMPANY, any value of Cust-ID determines a unique value of Name[9].

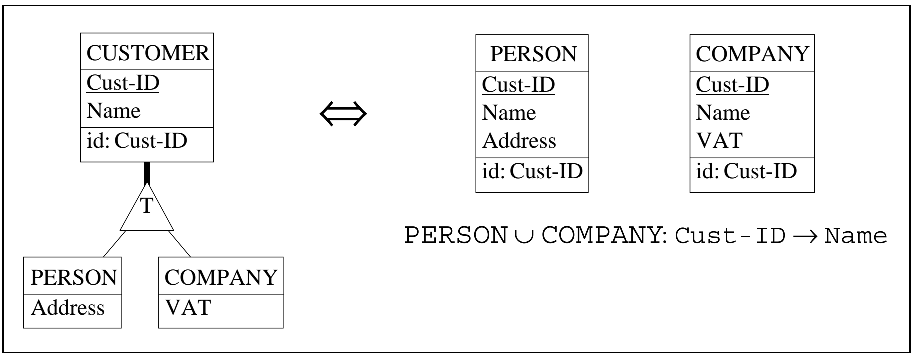


Figure 11.24 - Expressing the fact that if a PERSON entity and a COMPANY entity share the same value of Cust-ID, they also have the same value of Name.

Obviously, this technique should be avoided in ¬**D** IS-A hierarchies.

11.8 Elementary schema analysis

Identifying specific kinds constructs in a schema is a common task in many steps of database engineering, from conceptual analysis to logical and physical design and to reverse engineering. For instance, we can be interested in finding all the *entity types without identifiers* in a large schema. This tedious job can be automated with the help of the Global transformation assistant. To expreriment this, we open the schema LIBRARY/Conceptual, then we call

9. This property can be considered a special kind of *fonctional dependency*. Further detail on functional dependencies can be found in any textbook on databases, such as [Date 1999] or [Elmasri 2000] for instance.

the assistant. We select the problem Entity type - Missing id, then we choose the action Mark and we click on OK. We observe that the entity type AUTHOR, and only it, is marked[10].

To proceed to a deeper analysis of a schema we can build scripts. For instance, the script of Figure 11.25 mark the constructs of the current schema that are not SQL-compliant, i.e., the IS-A relations, the relationship types, the compound attributes and the multivalued attributes.

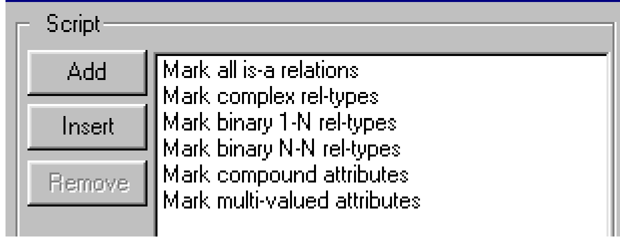


Figure 11.25 - A simple script that detects the most important non-relational constructs in an arbitrary schema.

Executing this script on the Conceptual schema results in the marked objects of Figure 11.22.

11.9 Advanced schema analysis

If we need to search a schema for more complex patterns, or if we want to check that a large schema is compliant with a definite model, the *Global transformation assistant* will quickly prove insufficient. In such situations, we will use the more powerful *Schema analysis assistant*.

We make the conceptual schema current, and we call the assistant by **Assist** / **Schema analysis**.

10. Marking objects consists in highlighting them as follows: we select the objects then we click on the button Mark in the Standard tool bar. *Marking* is a kind of permanent selection. To unmark objects, just click on the button Mark again.

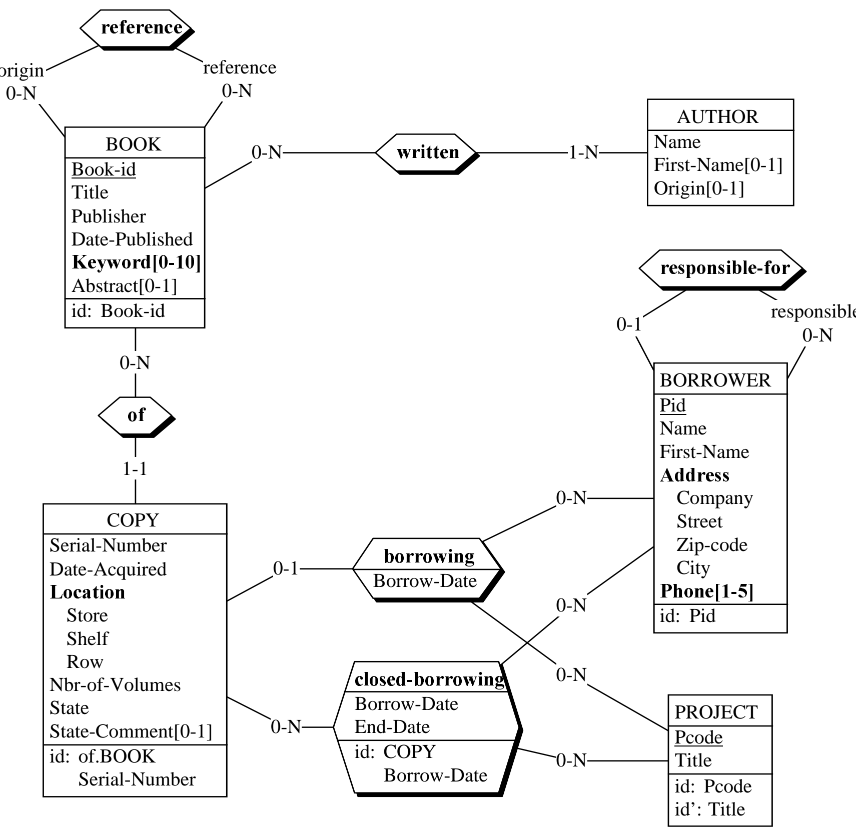


Figure 11.26 - Applying the validation script of the *Global transformation assistant* to highlight the non-relational constructs of the example schema.

The control panel of this assistant comprises five main sections (Figure 11.27):

Engine mode: defines the way rules are evaluated; in the Search mode, the engine searches the schema for structures that **obey** the constraints; in the Validate mode, the engine searches the schema for structures that **do not obey** the constraints.

Object: select the object type which the current constraint applies on.

Constraint: defines the constraint and its parameters.

Library: to build and use user-defined constraints.

Script manager: to build, update, save and load sets of constraints of arbitrary complexity.

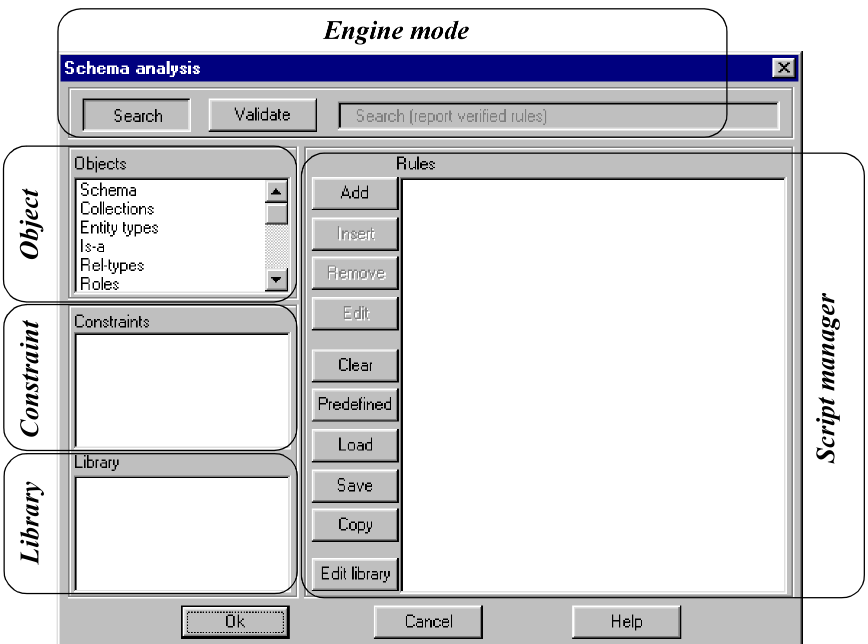


Figure 11.27 - The control panel of the Schema analysis assistant.

The *Search* mode

Now, let us define a first rule, describing the *entity types without identifiers*, as in the previous section. In the **Object** list, we select the item Entity type (Figure 11.27), and in the **Constraint** list we select the rule ID_per_ET, i.e., a constraint about the *number of identifiers per entity type*. We add this rule to the script area by clicking on the button Add. A new box opens, asking us the parameters of the rule. The latter defines the range (min max) of the number of identifiers. Since there must be none, we type the numbers "0 0" (Figure 11.28). Any consistent range can be typed, such as,

- 0 1 at most 1,
- 1 3 from 1 to 3,
- 1 N at least 1,

- 0 N any number, i.e., *no constraint*.

The definition of the rule can be obtained by clicking on the Help button.

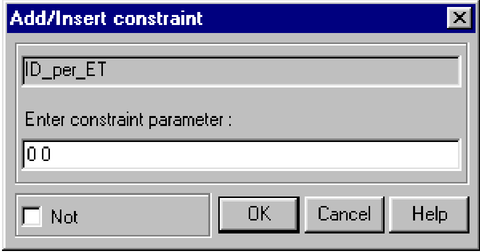


Figure 11.28 - Parameters of the rule ID_per_ET: describing entity types whose number of identifiers is between 0 and 0, i.e., with no identifiers.

Clicking on OK closes the definition of the constraint, which appears in the **Script area** (Figure 11.29).

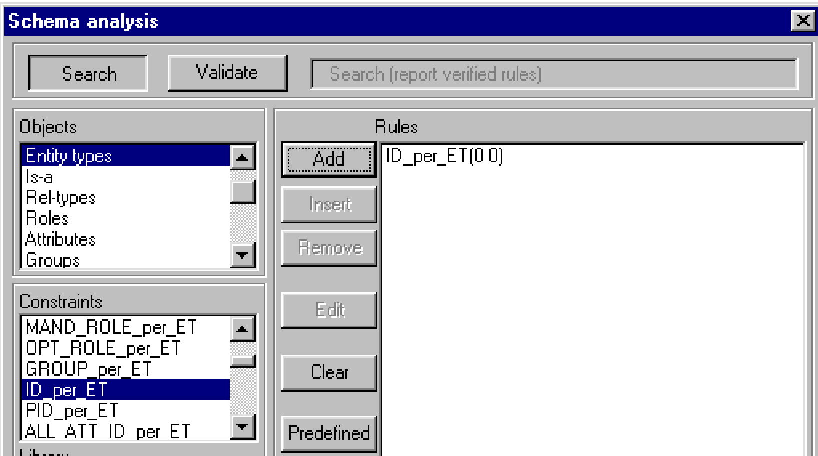


Figure 11.29 - The rule defining the *entity types with no identifiers* appears in the script area.

Now, clicking on the button OK in the control panel starts the search engine, which identifies all the instances of the pattern defined by the rule. The dia-

gnostic appears in a special box (Figure 11.30), which gives the rule and the constructs that satisfy this rule, here AUTHOR.

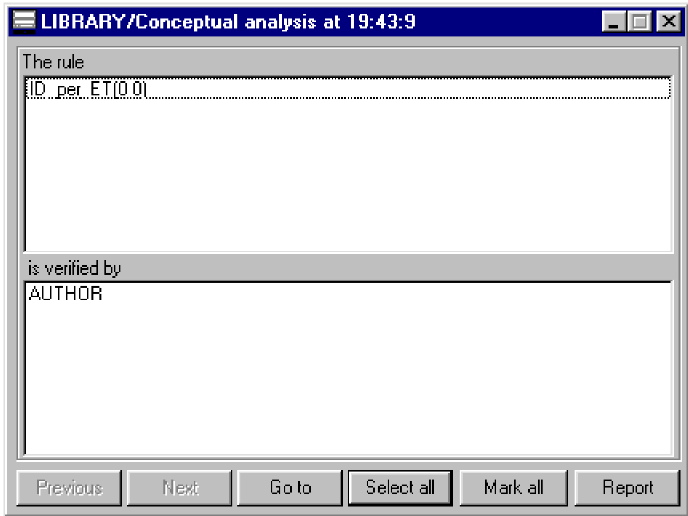


Figure 11.30 - The rule ID-per-ET(0 0) is satisfied by entity type AUTHOR.

Several rules can be inserted in the Script area, as in Figure 11.31, which orders the engine to search the schema for *non SQL-compliant* constructs: IS-A relations (ALL_ISA), rel-types (ALL_RT), multivalued attributes (with max cardinality from 2 to N) and compound attributes (with at least 1 sub-attribute). This list form a script which can be saved for further reuse.

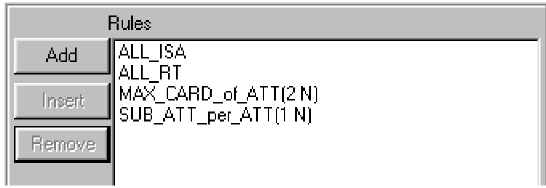


Figure 11.31 - Script defining four patterns that are *not SQL-compliant* (same effect as that of Figure 11.25).

Starting the search engine yields all the constructs that satisfy one of the rule of the script. The diagnostic box of Figure 11.32 shows the compound attributes of the schema. Through its control buttons we decide what to do with these constructs:

- button Previous — display the diagnostic for the previous rule,
- button Next — display the diagnostic for the next rule,
- button Go to — when an itemis selected in the list, display the corresponding object in the schema,
- button Select all — select all the objects mentioned in the list,
- button Mark all — mark all the objects mentioned in the list,
- button Report — print a report describing the objects mentioned in the list (Figure 11.33).

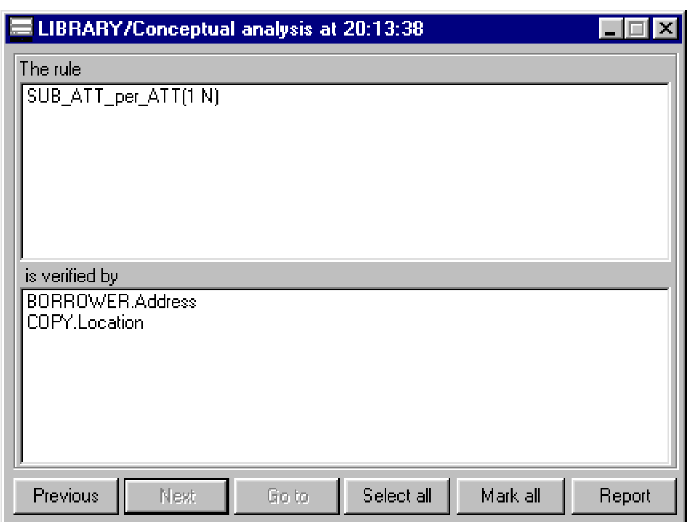


Figure 11.32 - The diagnostic box resulting from the analysis of the conceptual schema. It reports on the constructs that satisfy the rule `SUB_ATT_per_ATT(1 N)`, that describes compound attributes.

The *Validate* mode

In the Validate mode, the engine searches the schema for constructs that violate at least one rule. The rules now are interpreted as the properties that must hold in the current schema. in other words, the script defines a model.

```
The rule:
  ALL_RT
is verified by:
  borrowing
  closed-borrowing
  of
  reference
  responsible-for
  written

The rule:
  MAX_CARD_of_ATT(2 N)
is verified by:
  BOOK.Keyword
  BORROWER.Phone

The rule:
  SUB_ATT_per_ATT(1 N)
is verified by:
  BORROWER.Address
  COPY.Location
```

Figure 11.33 - Diagnostic report of the script of Figure 11.31 applied on the conceptual schema.

To illustrate the concept of model defined as a list of rules, we will examine the script defining the *relational model*. Before going into further detail, let us observe that, when defining the parameters of a rule, we can modify its value by using logical connectors, therefore writing complex rules from simpler ones:

- not *negates* the rule,
- and forms a logical *disjunction* with the previous rule,
- or forms a logical *conjunction* with the previous rule.

For example, the rule

```
ID_per_ET(1 N)
     and PID_per_ET(1 1)
  or ID_per_ET(0 0)
```

can be paraphrazed as

```
any entity type:
(has at least 1 id and has 1 primary id)
 or (has no id)
```

In other words, *any entity type, either has identifiers, and in this case, one of them must be primary, or it has no identifiers at all*, or, in better English, *if an entity type has some identifiers, one of them must be primary*.

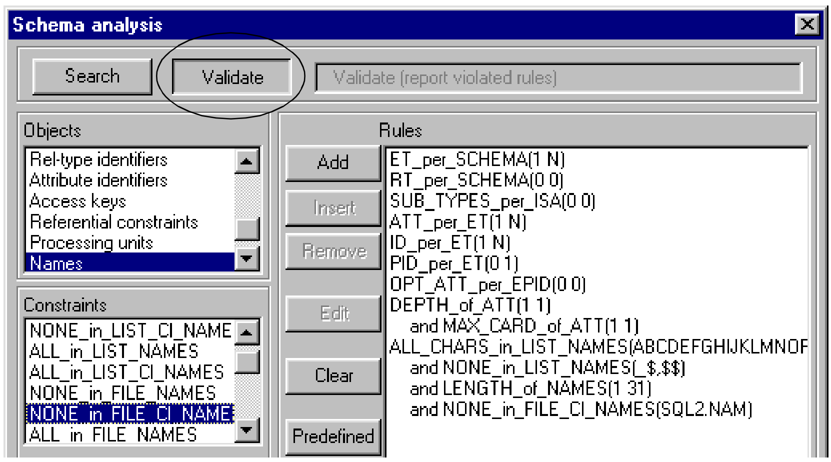


Figure 11.34 - A script that defines what rules all SQL-compliant schemas must satisfy. Note that the engine is in the *Validate* mode.

The script of Figure 11.34 defines what we could consider a good relational schema. Its interpretation is as follows:

- `ET_per_SCHEMA(1 N)`

  the schema must include at least one entity type;
- `RT_per_SCHEMA(0 0)`

  the schema must not include any rel-types;
- `SUB_TYPES_per_ISA(0 0)`

  no subtypes;
- `ATT_per_ET(1 N)`

  each entity type must include at least one attribute;
- `ID_per_ET(1 N)`

  each entity type must have at least one identifier
- `PID_per_ET(0 1)`

  no more than one primary id per entity type;
- `OPT_ATT_per_EPID(0 0)`

  no optional attributes on primary id of entity types;
- `DEPTH_of_ATT(1 1)`

all the attributes are at level 1;

- `and MAX_CARD_of_ATT(1 1)`

  ... and their maximum cardinality is 1 (single-valued);
- `ALL_CHARS_in_LIST_NAMES(ABCDEF...xyz012...89$_)`

  the names are made up of characters (upper or lowercase), digits, '`$`' and '`_`' (note that '...' stands for the actual characters);
- `and NONE_in_LIST_NAMES(_$,$$)`

  ... but their last character cannot be '`$`' nor '`_`';
- `and LENGTH_of_NAMES(1 31)`

  ... and they must be 1 to 31 character long;
- `and NONE_in_FILE_CI_NAMES(SQL2.NAM)`

  ... and they cannot appear in the list stored in the file `SQL2.nam`.

11.10Advanced schema transformation

The *Global transformation* assistant described in this lesson is quite intuitive and more than adequate for developing simple scripts. However, it cannot cope with complex problems, which require more sophisticated tools. The *Advanced global transformation* assistant is aimed at addressing this kind of problems. Compared with its little brother, this assistant does not provide a list of problems, but rather offers a *problem description facility* which is nothing else than the *Schema analysis* assistant! The action part is the list of schema transformation of DB-MAIN, plus some additional operations. In addition, it provides several control structures such as two kinds of loops and a library manager through which macro-rules and macro-actions can be developed.

Before discussing the scripting facility, we will experiment the building of a single action. Let us assume that we want to *transform the complex rel-types into entity types*.

First, we call the assistant by **Assist** / **Advanced global transformation** (Figure 11.35). Then, we proceed as follows.

1. in the *Primitive transformation* list, we select RT_to_ET;
2. we click on Add (or Insert) as in the first transformation assistant; a new box opens, asking us the definition the set of objects the transformation must apply on (Figure 11.36);

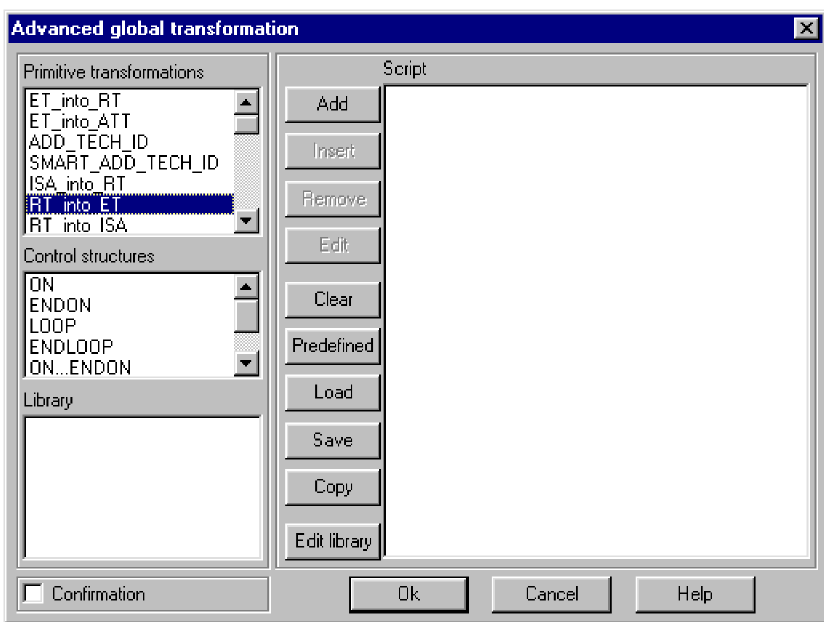


Figure 11.35 - The transformation *rel-type/entity type* is selected.

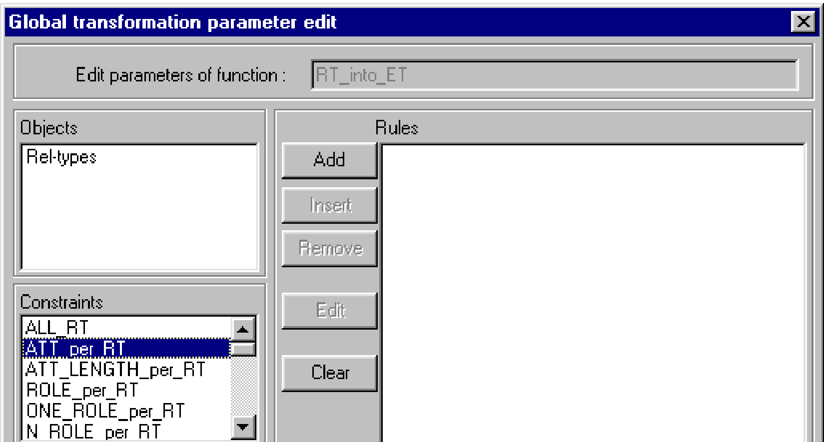


Figure 11.36 - The set of rel-type on which we want the transformation being applied is defined as a rule. We will build a rule by combining the elementary rule Att_per_RT and ROLE_per_RT.

3. we define the term *complex* as: *that has attributes or at least 3 roles*; therefore, we build the rule "ATT_per_RT(1 N) or ROLE_per_RT(3 N)" by selecting the corresponding constraints as in the *Schema analysis* assistant (Figure 11.36 and Figure 11.37);
4. we close the rule box (OK); the expression of the transformation is now complete (Figure 11.38);
5. we execute the transformation by clicking on the button OK.

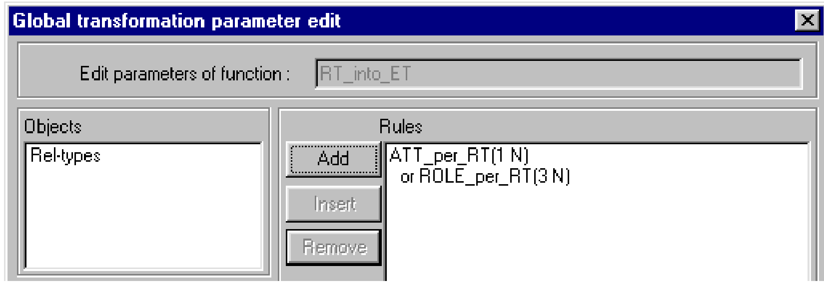


Figure 11.37 - The rule that defines complex rel-types.

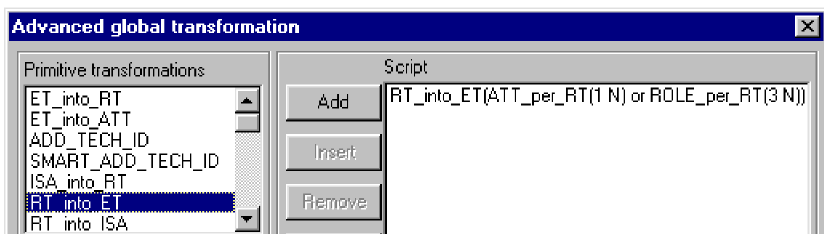


Figure 11.38 - The expression of the transformation is completed. It can be executed by clicking on OK.

Writing a script consists in building a series of such operations, as we did in the *Global transformation* assistant. The main addition is the loop control structure, the body of which comprises an arbitrary sequence of operations. At run time, the body is executed until the last execution did not change anything in the schema.

For instance, the fragment,

```
LOOP
   DISAGGREGATE
ENDLOOP
```

processes the current schema iteratively until no compound single-valued attributes can be disaggregated anymore, as in the example of Figure 11.39.

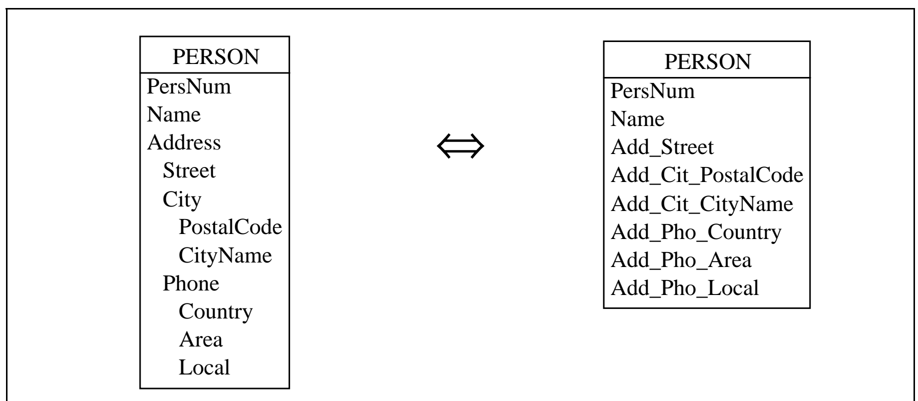


Figure 11.39 - Iterative decomposition of compound attributes.

To understand how all this works, we will develop a small script that transforms conceptual schemas into SQL-compliant schemas.

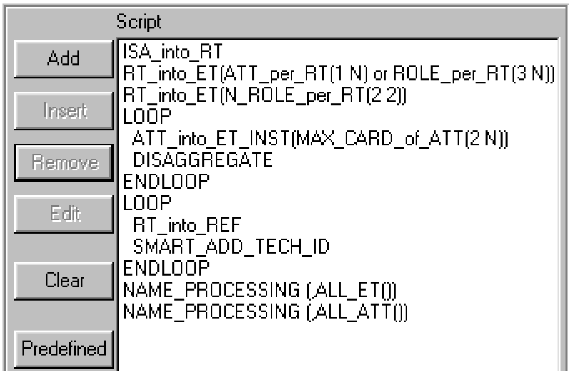


Figure 11.40 - A transformation script that can produce a SQL-compliant schema from (almost) any entity relationship schema.

We follow the transformation plan of Figure 11.5 and Figure 11.15, which translates into the script of Figure 11.40.

Here follows a short description of the statements the script is made up of.

- `ISA_into_RT`
  transform all IS-A relations into one-to-one rel-types;
- `RT_into_ET(ATT_per_RT(1 N) or ROLE_per_RT(3 N))`
  transform complex rel-types (with attributes or at least 3 roles) into entity types;
- `RT_into_ET(N_ROLE_per_RT(2 2))`
  transform many-to-many rel-types (with 2 roles of type many) into entity types;
- `LOOP`
  start a loop that processes complex attributes;
- `ATT_into_ET_INST(MAX_CARD_of_ATT(2 N))`
  transform level 1 multivalued attributes (with max-card at least 2) into entity types through instance representation;
- `DISAGGREGATE`
  disaggregate level 1 single-valued attributes;
- `ENDLOOP`
  close the loop; the loop body is run until no attributes can be transformed;
- `LOOP`
  start a new loop that process simple rel-types;
- `RT_into_REF`
  transform all rel-types into reference attributes (foreign keys); the operation is rerun until no rel-type can be transformed;
- `SMART_ADD_TECH_ID`
  add a technical id to all the entity type that need one in order to allow rel-types to be transformed (= smart)
- `ENDLOOP`
  close the loop and rerun the body until no object can be transformed;
- `NAME_PROCESSING(ALL_ET())`
  translate the names of the entity types;

- `NAME_PROCESSING(ALL_ATT())`
  translate the names of the attributes.

This script can be saved for further reuse. The Predefined button provides a list of built-in scripts that can be used *off-the-shelves*, or as the basis for developing new scripts. Worth being examined carefully.

Summary of Lesson 11

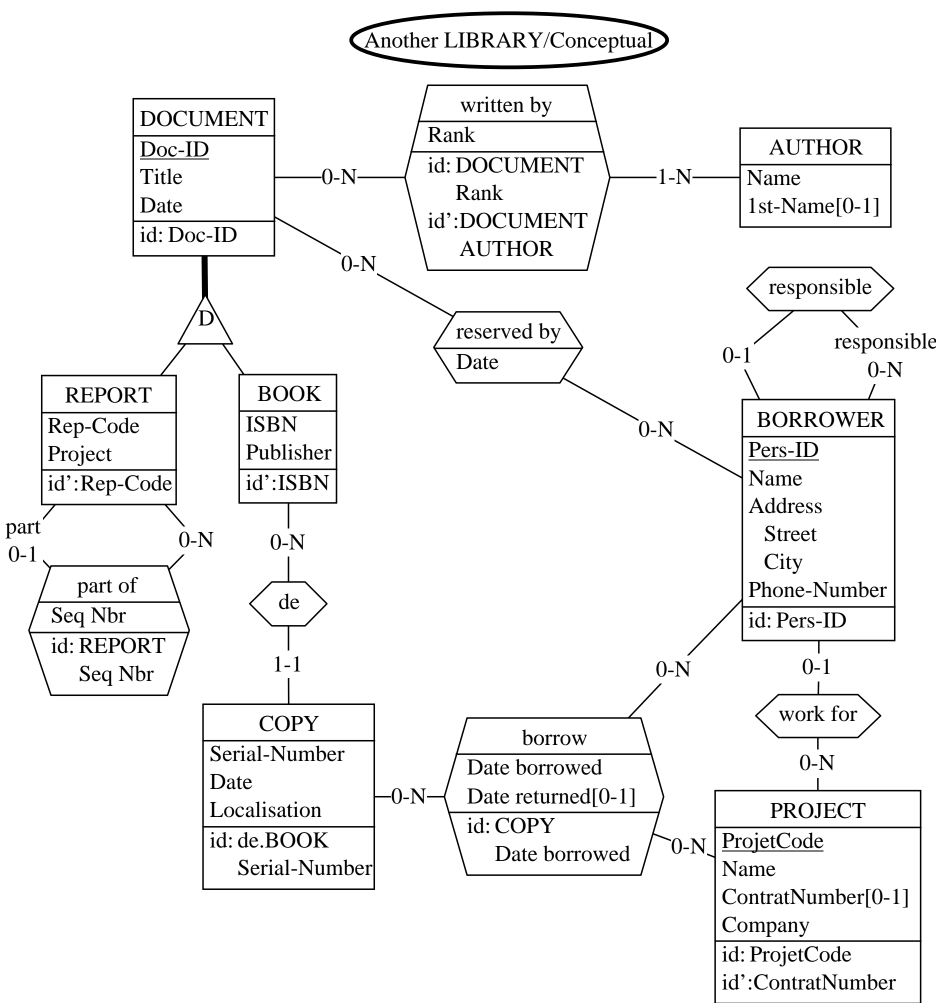
- In this lesson, we have studied new notions:
  - transformation plan;
  - assistants, and particularly the *Global transformation* and *Schema analysis* assistants;
  - problem/solution statements and scripts;

- We have also learnt how
  - to transform an *IS-A* relation into *one-to-one* rel-types:

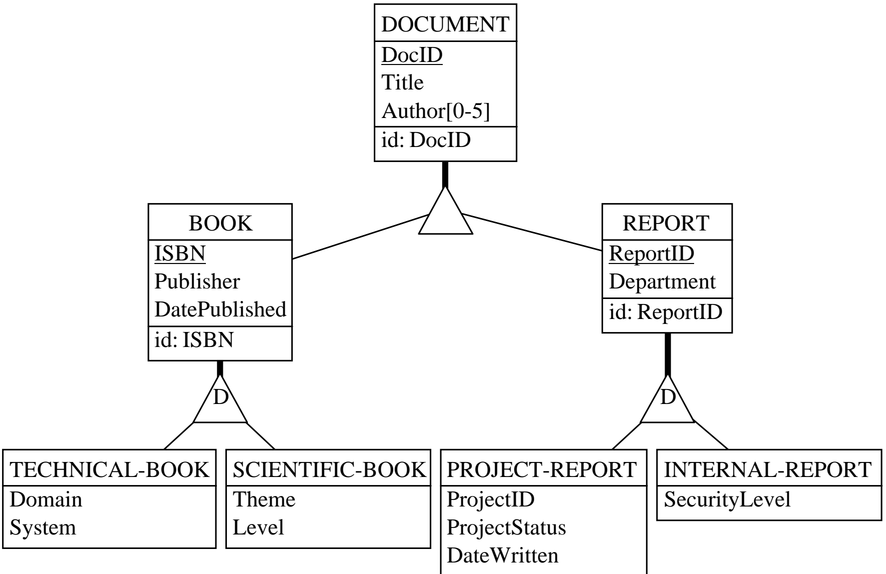
    Transform / Entity type / Is-a -> rel-type
  - to derive other equivalent structures for IS-A relations
  - to use the *Problem solver* of the Global transformation assistant
  - to use the *Schema analysis* assistant
  - to use the *Advanced global transformation* assistant
  - to build and manage scripts

Exercises for Lesson 11

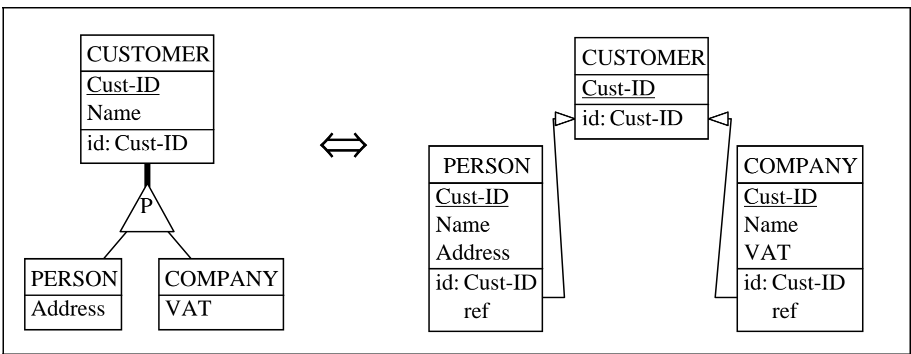
11.1 Transform the following conceptual schema into a SQL-compliant schema. Try several translations of the IS-A relation.



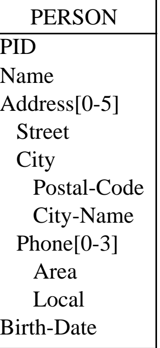
11.2 Transform the following schema into SQL-compliant structures.



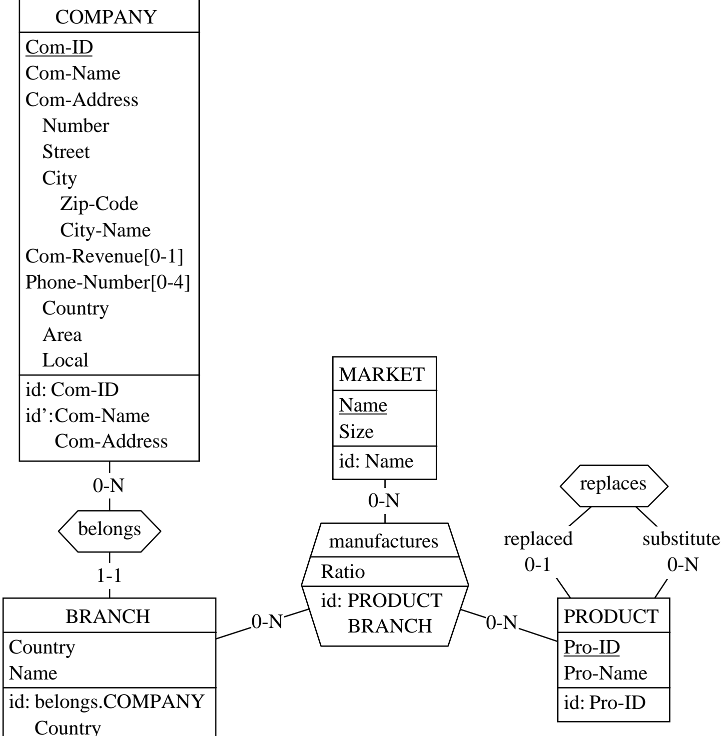
11.3 To help solve the difficulties of managing the IS-A representation through the downward inheritance technique, we could propose an implementation based on the following pattern. Try to justify to what extend this improve the management of the **D** constraint. Develop a set of triggers to automatically manage this constraint.



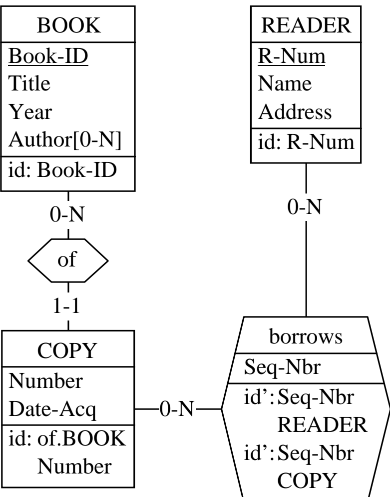
11.4 Apply the transformation plan we have built in this lesson to the following conceptual schema. Analyze the result carefully.



11.5 Same exercise with this schema:



11.6 ... and with this one:



11.7 Define a transformation plan to express ORM schemas into Entity-relationship schemas (see *Exercises* of Lesson 8). Write, check and save a script that implements this transformation plan.

11.8 Choose a record structure you are acquainted with, such as Pascal, C or COBOL. Design and implement in the *Schema analysis* assistant a set of rules that can be used to validate any schema against this structure.

11.9 Design and implement a transformation plan that produces a record structure (Pascal, C or COBOL) from any conceptual schema.

Lesson 12

Physical design

Objective

This is the last lesson dedicated to the LIBRARY case study. It will introduce to the derivation of the **physical schema**, i.e., the schema which specifies not only the logical structures, but also technical characteristics of the database such as the indexes and the files in which table rows are stored. In addition, it examines the translation rules into the DBMS data description language.

12.1 Starting Lesson 12

We start DB-MAIN and we open the project `logical-10` (not `logical-11`!) which includes the conceptual schema of the database in project, as well as the final version of the logical schema. We save it as `LIBRARY`.

12.2 What is a physical schema?

There are several interpretations of the concept of **physical schema** of a database. Historically speaking, the physical schema was first understood as the collection of the technical characteristics of the implemented data structures: index structures, buffer size, page size, free space at loading time, clustering, pointers, and the like.

We will give this concept a more recent interpretation: the physical schema is the whole collection of specifications one have to give the DBMS in order to get an operational database. The physical schema is thus made up of the logical schema + the technical characteristics and parameters. From the practical point of view, a physical schema must be expressed into the specific *data definition language* (or DDL) of the target DBMS.

The logical schema depends on the model of a family of DBMS: for instance, a logical model is relational (what we called SQL-compliant), but a physical schema is compliant with a specific DBMS of this family, such as ORACLE V7 or V8, DB2, Informix, SQL Server or SYBASE. In the same way, from a CODASYL logical schema one can derive an IDS-2 (Bull) physical schema, or an UDS (Siemens) schema or an IDMS (CA) schema. This organization is summarized in the following project structure, in which a conceptual schema has been translated into an SQL logical model and into a CODASYL logical schema, and each of them has in turn been translated in a series of physical schemas. In addition, each physical schema has been expressed into the DDL of its DBMS. This hypothetical project covers six physical versions of the same conceptual schema.

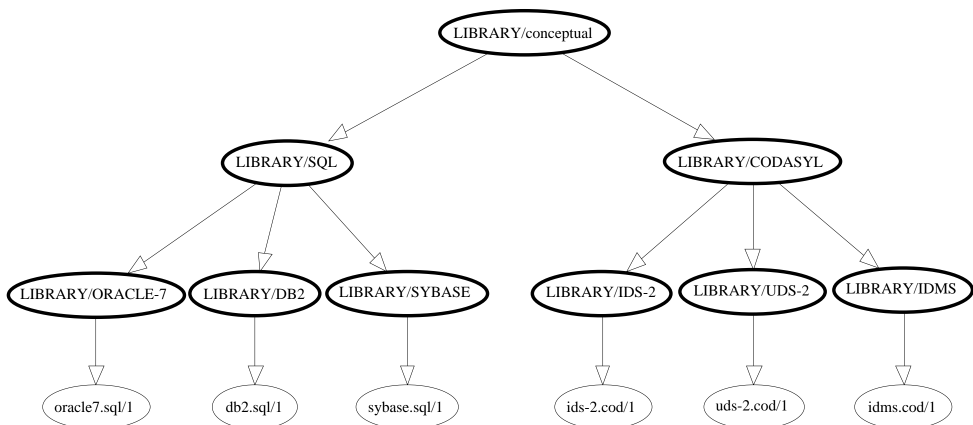


Figure 12.1 - A multi-target project: the same conceptual schema has been transformed into a relational schema and into a CODASYL schema, then each of them has been implemented into several DBMS and translated into DDL programs.

12.3 And what about physical design?

Physical design is a complex and highly knowledge-based activity. Indeed, developing a schema which satisfies such conflicting criteria as time performance, space minimization (core memory and disk), smooth evolution, ease of maintenance, portability, modularity, ease of exploitation, requires much technical expertise, and obviously is not a job for the novice analyst.

Of course, in the limited scope of an introduction to database design, we cannot go into too detailed a development. So, we will propose a very simplified approach, quite sufficient to grasp the concept, but a bit too superficial to get the real taste.

We will consider two phases in physical design. The first one consists in developing an abstract physical schema by augmenting the logical schema with technical specifications. Through the second phase, this physical schema is translated into the DDL of the DBMS.

12.4 Building the physical schema of a database

We will proposed a simplified procedure which would provide acceptable performances in *not-too-demanding* applications. The specific features which will transform a logical schema into a physical schema are: the *access keys* (index) and the *entity collections* (files). A little touch of optimization will also be discussed: discarding redundant access keys.

The access keys (indexes)

The concept of access key has been presented in Lesson 5 (Section 5.7). An access key represents any technical data structure that provides quick and selective access to data, therefore avoiding time-consuming sequential access. In the relational database technology, access keys are implemented as *index*, *bit-map* or *hash* organization.

Deciding which columns, or column combinations, should be access keys is a complex task based on a careful analysis of the application programs requirements. Since we have no information on these requirements (not even on the programs themselves), we can only make reasonable assumptions on them.

For instance, it is not completely unrealistic to suppose that each **identifier** should be an access key as well. Indeed, an identifier often is a preferred selection criterion, for instance to designate a specific object, or to carry out joins. In addition, inserting a new entity (i.e., a row in a table) requires checking the non-existence of the identifier value in the entity set (i.e., the table).

Another reasonable hypothesis concerns the **foreign keys**. Indeed, each of them derives from a rel-type, which represent an outstanding semantic structure. Most probably, many application programs should use this structure to retrieve data. We thus make each foreign key an access key.

Other access keys can be added if we think they will be strongly useful to get better access time. However, such decision cannot be fully justified without knowledge on the data applications.

An interesting feature of relational DBMS is that they allow dynamically creating and dropping an index during the life of the database, and not only at the definition stage. Therefore, if, later on, we observe that the usage of an index is lower than expected, we can discard it without restructuring the database. Similarly, if we think than a formerly unplanned index would have been useful, we can add it dynamically. So, an initial error in index definition is harmless.

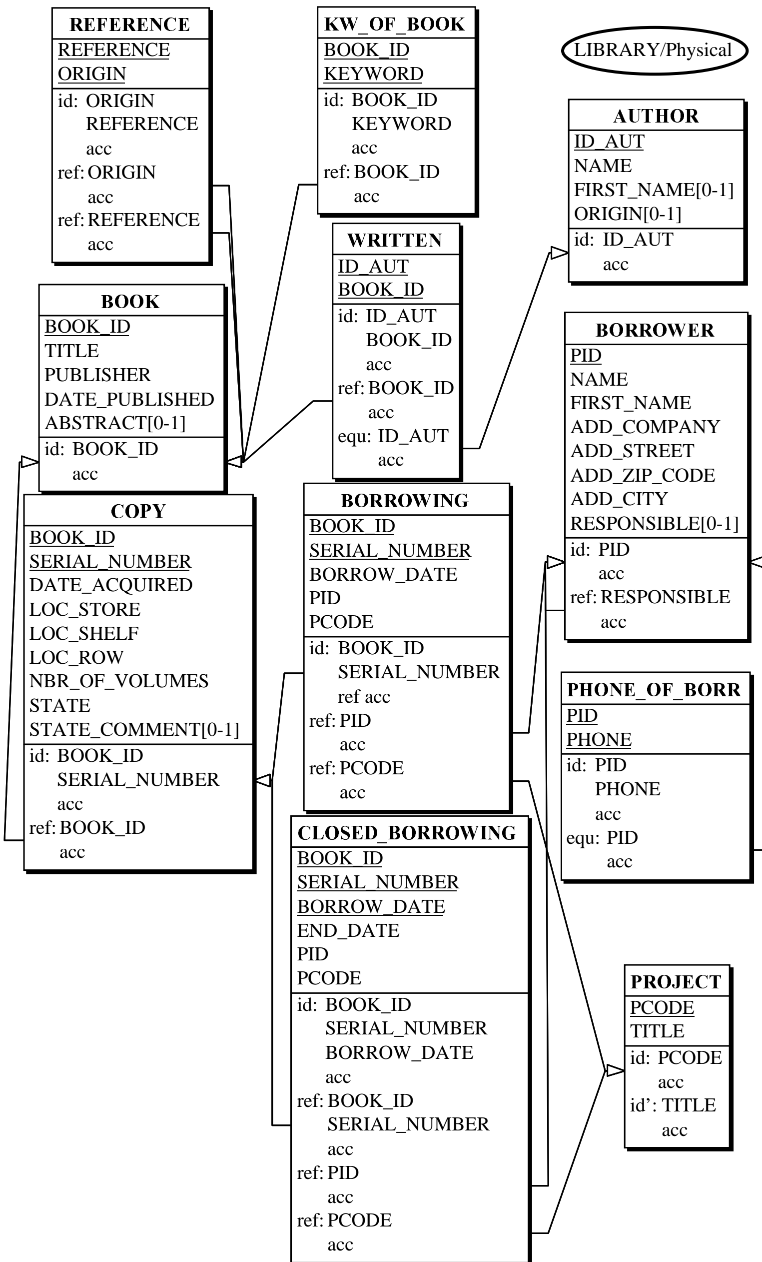


Figure 12.2 - Assigning *access keys* (indexes) to identifiers and foreign keys.

The schema of Figure 12.2 is the first version of the physical schema. It has been obtained as follows:

- we select the `LIBRARY/Logical` schema,
- we copy it under the name/version: `LIBRARY/Physical`, and we open it,
- we open all the identifiers and reference groups of this schema, and click the access key button.

Each id, id', ref and equ group has been complemented with the specification acc (for access key).

12.5 Redundant access keys

Though optimizing schemas and physical tuning are not addressed in this volume, we can apply the popular rule about the indexes of a table or file shortly mentioned in Section 5.7. The rule is well-known by COBOL programmers, and comes as follows:

Rule: if `X1` is a sorted index, if `X2` is another index, and if the fields of `X2` form a **prefix** of the fields of `X1`, then **`X2` can be dropped**.

Example: the records of a file comprises fields `A1`, `A2`, `A3`, `A4`; the file has three indexes, based on <`A1`,`A2`,`A3`>, <`A1`,`A2`> and <`A1`>; the indexes are implemented by any sort of tree-based techniques (ISAM, B-tree, etc); in such a situation, the indexes <`A1`,`A2`> and <`A1`> can be discarded because the DBMS can use the full index to simulate the other two.

Comments: an index based on <`A2`,`A3`>, <`A3`>, or even on <`A2`,`A1`> must be kept; in addition, if the implementation of the index is based on hashing techniques, then the rule does not applies.

We can adapt this rule to the logical model used in DB-MAIN:

Rule: if access key `X2` is a prefix of access key `X1`, then `X2` can be discarded (Figure 12.3).

In this case study, we will suppose that the index implementation techniques satisfy the *sorted* hypothesis.

The current version of the physical schema includes several patterns of this kind. Let us examine a single example (Figure 12.4).

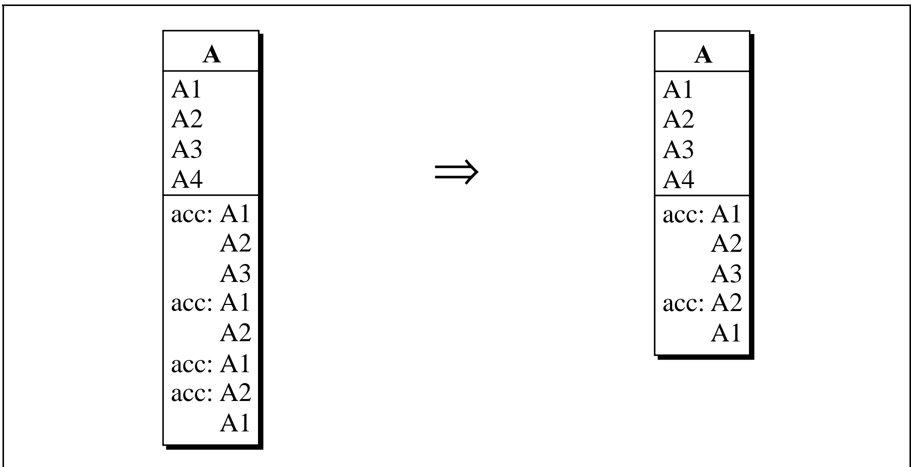


Figure 12.3 - Removing prefix access keys.

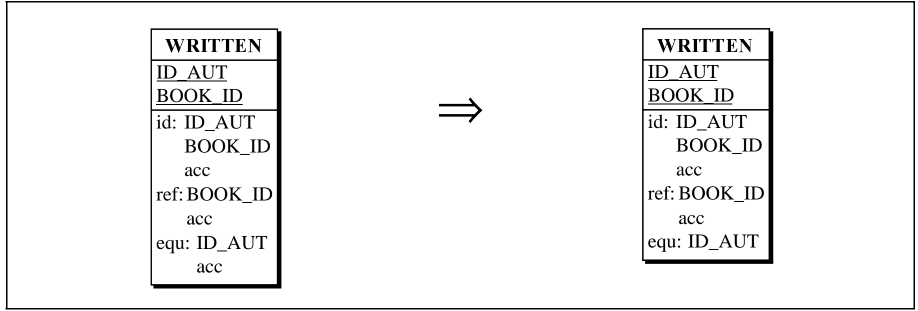


Figure 12.4 - Minimizing the number of access keys of `WRITTEN`.

Some situations do not comply with this pattern, though shuffling the component of the access keys can make prefix access key appear. Let us assume that the table `PHONE_OF_BORR` has been given the identifier and the foreign key of Figure 12.5/left. Obviously, no access key is a prefix of the other. However, swapping the components of the identifier makes such a pattern appear (Figure 12.5/right). Now we can minimize the access keys:

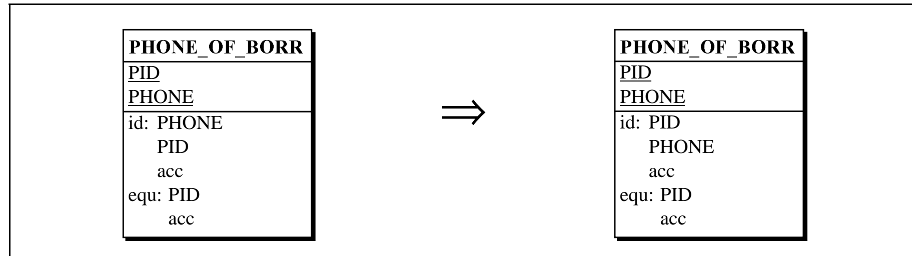


Figure 12.5 - Shuffling the components of access keys can make prefix access keys appear.

Other entity types can be restructured in this way, in order to drop unnecessary access keys. Be careful however, when you swap the components of an access key which is also a foreign key, or a referenced id, you must preserve the order of the components at the other side.

The final physical schema is presented in Figure 12.6.

The entity collections (files)

Entity collection is a general name for such things as *files*, *datasets*, *areas*, *realms*, *table spaces*, *DBspaces*, and any other physical stores (see Lesson 5, Section 5.8). In a relational DBMS, a file is often called *space*, *DBspace* or *table space*[1].

The designer must specify which files (*collections*) are available, and in which file(s) the rows of each table will be stored. There are many reasons for which this assignation can induce good or bad database behaviour, but reasoning on this is beyond the scope of this volume. We will only define the files (i.e., the collections), and specify which tables (i.e., the entity types) will be stored in these files.

We decide to split the database into two logical subparts:

- the ***book*** part, including the tables `BOOK`, `COPY`, `REFERENCE`, `WRITTEN`, `AUTHOR` and `KW_OF_BOOK`,
- the ***borrowing*** part, which includes the other tables: `BORROWER`, `PROJECT`, `BORROWING`, `CLOSED_BORROWING` and `PHONE_OF_BORR`.

1. In some systems, the organization can even be more complex: logical files are mapped to physical files.

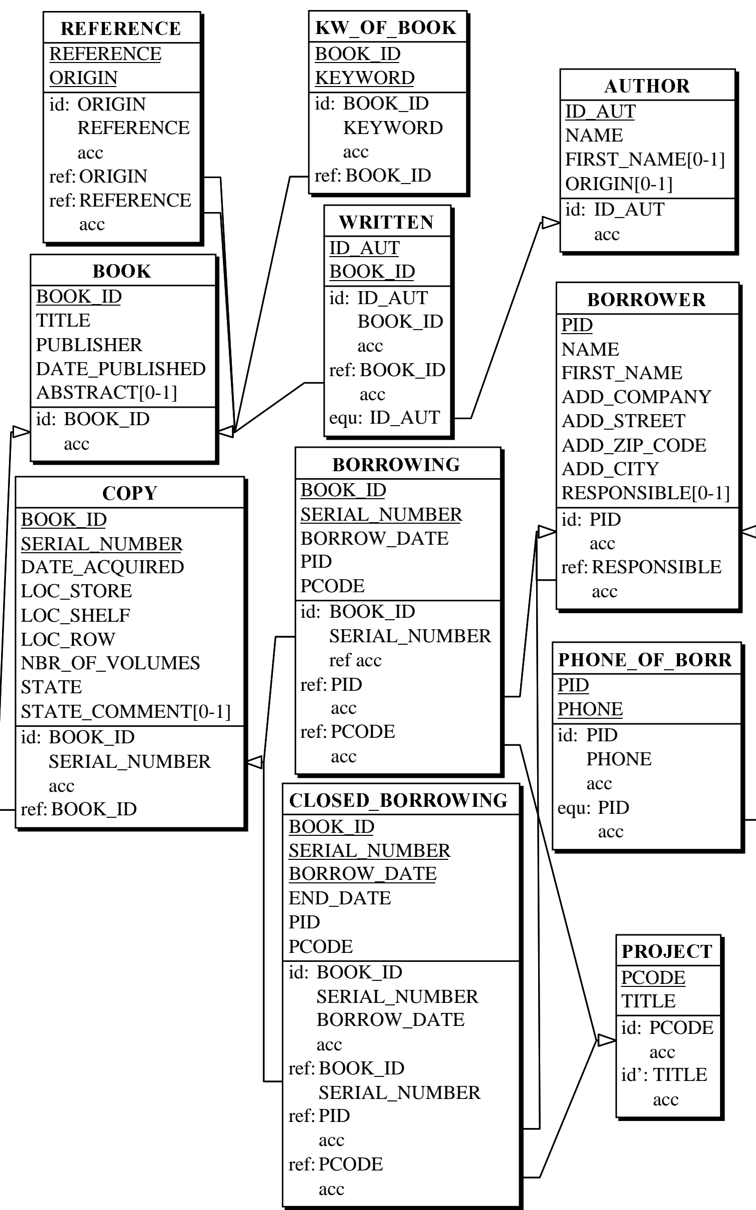


Figure 12.6 - Removing the prefix access keys.

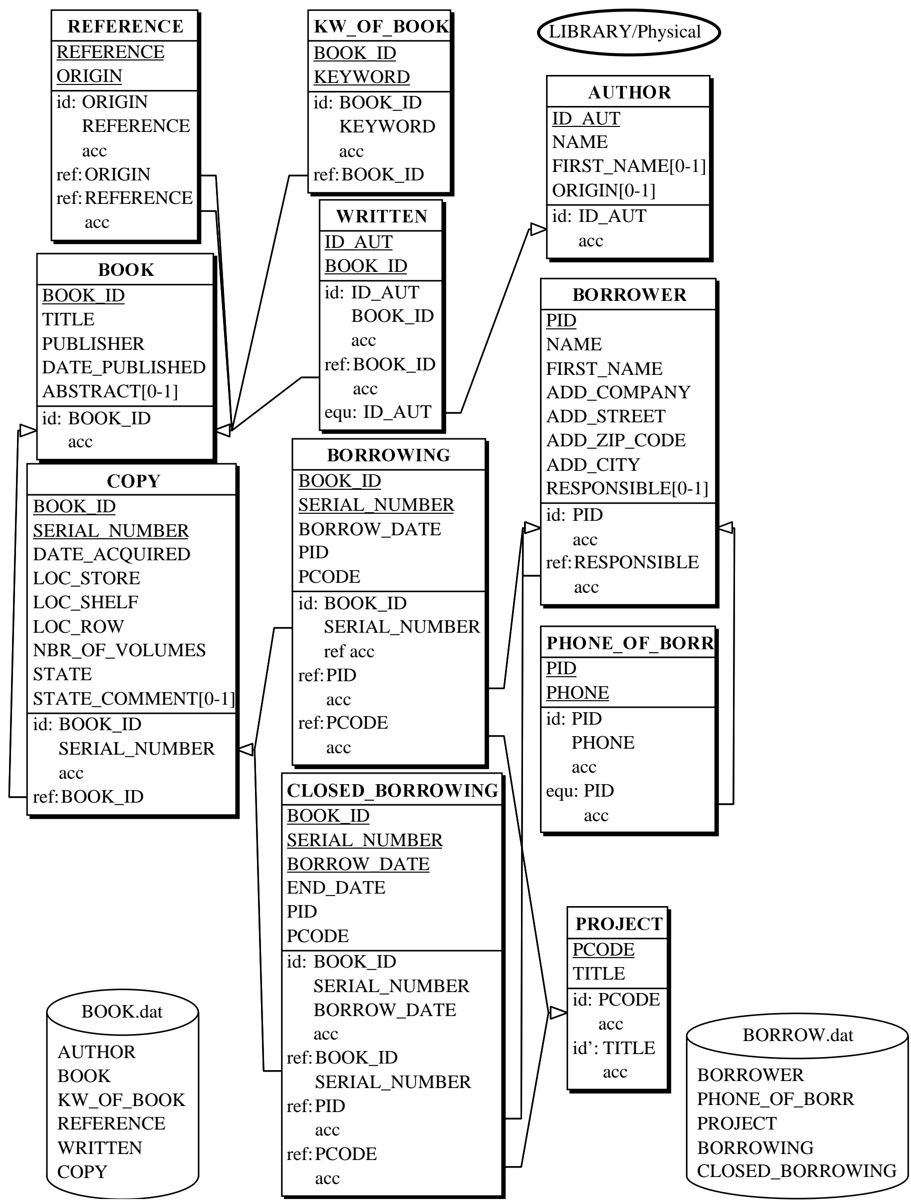


Figure 12.7 - The final physical schema. The tables have been assigned to *collections* (storage spaces).

We also decide to assign the data of each subpart to a specific entity collection, namely `BOOK.dat` and `BORROW.dat`. So, we create two collections (command **New** / **Collection**, or through the button), and we assign to each of them the corresponding tables. The schema is shown in Figure 12.7.

12.6 The TECH descriptions

The technical description (the one which is available from the button TECH in each object property box) is the perfect place to write the recommandations we find useful to transmit to, say, the programmer, or the database manager. For instance, we can specify DBMS-dependent physical parameters.

12.7 Generating the DDL schema

This coding activity consists in writing the DDL expression of each construct of the physical schema. In general, a set of coding rules must be defined for each DBMS. Moreover, each company, each methodology, and even each analyst can have its own coding style. For instance, in relational databases, an identifier can be coded as a *primary key*, as a *unique constraint*, as a *unique index*, as a *check predicate* or as a *trigger*. Declaring constraints in the table declaration or as an alter table, as well as naming, or not, constraints, also are a matter of style.

Generating the DDL text of a database structure is thus highly context-dependent. Therefore, we can only propose simple and intuitive coding rules that should be adequate in most circumstances[2].

Instead of giving a comprehensive, and therefore tedious, list of coding rules, we propose the following translation of the physical schema.

---

2. Specific coding styles can be defined thanks to customized generators which can be developed by the analyst (or the methodologist) in the *Voyager 2* language, the external development language of DB-MAIN, or through the SQL generator of DB-MAIN, which allows the analyst to select the coding style of each type of constaint, or even of each constraint.

```
--------- DB and DBSPACES ---------
create database LIBRARY;

create dbspace BORROW_dat;
create dbspace BOOK_dat;
```

```
--------------- TABLES ---------------
create table AUTHOR (
     ID_AUT char(10) not null,
     NAME char(30) not null,
     FIRST_NAME char(30),
     ORIGIN char(30),
     primary key (ID_AUT))
     in BOOK_dat;

create table BOOK (
     BOOK_ID numeric(6) not null,
     TITLE char(30) not null,
     PUBLISHER char(40) not null,
     DATE_PUBLISHED date not null,
     ABSTRACT char(80),
     primary key (BOOK_ID))
     in BOOK_dat;
create table BORROWER (
     PID char(6) not null,
     NAME char(30) not null,
     FIRST_NAME char(30) not null,
     ADD_COMPANY char(40) not null,
     ADD_STREET char(40) not null,
     ADD_ZIP_CODE numeric(4) not null,
     ADD_CITY char(40) not null,
     RESPONSIBLE char(6),
     primary key (PID))
     in BORROW_dat;

create table BORROWING (
     BOOK_ID numeric(6) not null,
     SERIAL_NUMBER numeric(6) not null,
     BORROW_DATE date not null,
     PID char(6) not null,
     PCODE char(6) not null,
     primary key (BOOK_ID, SERIAL_NUMBER))
     in BORROW_dat;
```

```
create table CLOSED_BORROWING (
     BOOK_ID numeric(6) not null,
     SERIAL_NUMBER numeric(6) not null,
     BORROW_DATE date not null,
     END_DATE date not null,
     PID char(6) not null,
     PCODE char(6) not null,
     primary key (BOOK_ID, SERIAL_NUMBER, BORROW_DATE))
     in BORROW_dat;

create table COPY (
     BOOK_ID numeric(6) not null,
     SERIAL_NUMBER numeric(6) not null,
     DATE_ACQUIRED date not null,
     LOC_STORE numeric(2) not null,
     LOC_SHELF numeric(2) not null,
     LOC_ROW numeric(2) not null,
     NBR_OF_VOLUMES numeric(3) not null,
     STATE char(10) not null,
     STATE_COMMENT char(80),
     primary key (BOOK_ID, SERIAL_NUMBER))
     in BOOK_dat;

create table KW_OF_BOOK (
     BOOK_ID numeric(6) not null,
     KEYWORD char(30) not null,
     primary key (BOOK_ID, KEYWORD))
     in BOOK_dat;
create table PHONE_OF_BORR (
     PID char(6) not null,
     PHONE numeric(10) not null,
     primary key (PID, PHONE))
     in BORROW_dat;

create table PROJECT (
     PCODE char(6) not null,
     TITLE char(30) not null,
     primary key (PCODE),
     unique (TITLE))
     in BORROW_dat;

create table REFERENCE (
     REFERENCE numeric(6) not null,
     ORIGIN numeric(6) not null,
     primary key (ORIGIN, REFERENCE))
     in BOOK_dat;

create table WRITTEN (
     ID_AUT char(10) not null,
     BOOK_ID numeric(6) not null,
     primary key (ID_AUT, BOOK_ID))
     in BOOK_dat;
```

```
------------- Checks for EQU reference attributes ----------
alter table AUTHOR add constraint
     check(exists(select * from WRITTEN
                  where WRITTEN.ID_AUT = ID_AUT));
alter table BORROWER add constraint
     check(exists(select * from PHONE_OF_BORR
                  where PHONE_OF_BOR.PID = PID));
```

```
---------------- foreign keys ----------------
alter table BORROWER add constraint FKRESPONSIBLE_FOR
     foreign key (RESPONSIBLE) references BORROWER;
alter table BORROWING add constraint FKBOR_COP
     foreign key (BOOK_ID, SERIAL_NUMBER) references COPY;
alter table BORROWING add constraint FKBOR_BOR
     foreign key (PID) references BORROWER;
alter table BORROWING add constraint FKBOR_PRO
     foreign key (PCODE) references PROJECT;
alter table CLOSED_BORROWING add constraint FKCLO_COP
     foreign key (BOOK_ID, SERIAL_NUMBER) references COPY;
alter table CLOSED_BORROWING add constraint FKCLO_BOR
     foreign key (PID) references BORROWER;
alter table CLOSED_BORROWING add constraint FKCLO_PRO
     foreign key (PCODE) references PROJECT;
alter table COPY add constraint FKOF
     foreign key (BOOK_ID) references BOOK;
alter table KW_OF_BOOK add constraint FKBOO_KW_
     foreign key (BOOK_ID) references BOOK;
alter table PHONE_OF_BORR add constraint FKBOR_PHO
     foreign key (PID) references BORROWER;
alter table REFERENCE add constraint FKORIGIN
     foreign key (ORIGIN) references BOOK;
alter table REFERENCE add constraint FKREFERENCE
     foreign key (REFERENCE) references BOOK;
alter table WRITTEN add constraint FKWRI_BOO
     foreign key (BOOK_ID) references BOOK;
alter table WRITTEN add constraint FKWRI_AUT
     foreign key (ID_AUT) references AUTHOR;
```

```
------------- INDEXES -------------
create unique index ID on AUTHOR (ID_AUT);
create unique index ID_BOOK on BOOK (BOOK_ID);
create unique index ID_BORROWER on BORROWER (PID);
create index FKRESPONSIBLE_FOR on BORROWER (RESPONSIBLE);
create unique index FKBOR_COP on BORROWING (BOOK_ID, SERIAL_NUMBER);
create index FKBOR_BOR on BORROWING (PID);
create index FKBOR_PRO on BORROWING (PCODE);
create unique index ID_CLOSED_BORROWING
       on CLOSED_BORROWING (BOOK_ID, SERIAL_NUMBER, BORROW_DATE);
create index FKCLO_BOR on CLOSED_BORROWING (PID);
create index FKCLO_PRO on CLOSED_BORROWING (PCODE);
```

```
create unique index ID_COPY on COPY (BOOK_ID, SERIAL_NUMBER);
create unique index IDKW_OF_BOOK on KW_OF_BOO (BOOK_ID, KEYWORD);
create unique index IDPHONE on PHONE_OF_BOR (PID, PHONE);
create index FKBOR_PHO on PHONE_OF_BOR (PID);
create unique index ID_PROJECT on PROJECT (PCODE);
create unique index ID_PROJECT_2 on PROJECT (TITLE);
create unique index IDREFERENCE on REFERENCE (ORIGIN, REFERENCE);
create index FKREFERENCE on REFERENCE (REFERENCE);
create unique index IDWRITTEN on WRITTEN (ID_AUT, BOOK_ID);
create index FKWRI_BOO on WRITTEN (BOOK_ID);
```

The complete project that we are developing since Lesson 6 includes four products, as illustrated in 12.8.

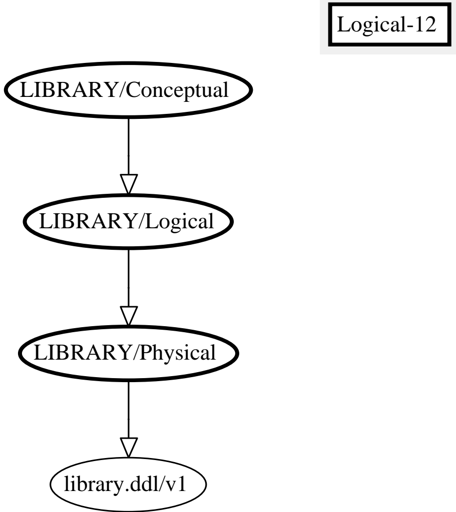


Figure 12.8 - The product hierarchy of the project.

12.8 Getting help from DB-MAIN

DB-MAIN includes specific functions to help building physical schemas and generating DDL texts. The first function is the relational model transformation, already used in Leson 3, while the others are available through the *Global transformation* assistants, and can be included into scripts.

The Relational model transformation

This operator comprises a built-in transformation plan which translates the current (conceptual or logical) schema into a physical schema. This tool follows a set of rules similar to those which has been described in lessons 11 and 12. It can be called by the command **Transform** / **Relational model**.

The Global transformation assistant

Besides the *Problem/solution* statements presented in Lesson 11, this assistant includes other functions to process a schema more quickly than through individual transformations. In addition, these statements can be included into a script, in order to automatically build physical schemas.

We mention the main statements useful in physical design. They are specified by their expression in the script area:

- **Make access key from id. or ref.**

  All the identifiers and foreign keys are made access keys.
- **Add tech. id. when id. > 1 component**
- **Add tech. id. when id. > 2 components**
- **Add tech. id. when id. > 3 components**

  If the primary identifier is made up of more than 1 (or 2 or 3) component(s), replace it by a technical identifier. This optimization technique can simplify a schema by including simple and short primary identifiers and foreign keys instead of complex ones.
- **Remove prefix access keys**

  Remove any access key which is the prefix of another one.
- **Rename all groups**

  Replace the group names with systematic names. Group names will be assigned to indexes and constraints for instance.
- **Generate X**

  Generate the DDL text for the current schema according to style X (to be selected).

The predefined script called *Pseudo-relational* is an example of such *logical-physical* script. Other scripts exist, such as *Pseudo-CODASYL* and *Pseudo-COBOL*, for instance, that can produce acceptable CODASYL and COBOL

file structures. These scripts are not quite comprehensive, and may fail for some complex schemas, hence the qualifier *pseudo*.

12.9 Quitting the lesson

We can now quit DB-MAIN. The modified project can be saved with the name `Library.lun`.

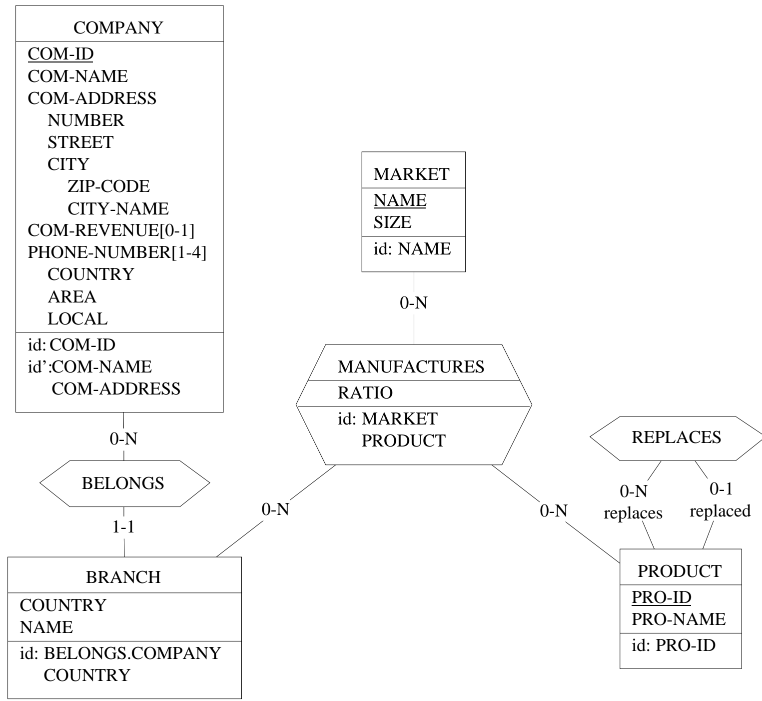
Summary of Lesson 12

- In this lesson, we have studied new notions:
  - physical design
  - prefix access key

- We have also have learnt to
  - choose and define access keys
  - to write scripts for physical design

Exercises for Lesson 12

12.1 Propose a relational physical version for this conceptual schema:



12.2 **Reverse Engineering**. Design a transformation plan to recover the conceptual schema from any relational physical schema. Write the corresponding script in one of the Global transformation assistant. Test it on the schema LIBRARY/Physical we developed in this lesson. Compare your solution with the predefined scripts of the assistants (this operation is called reverse engineering).

12.3 **Reverse Engineering**. Open a new project and import the SQL text we have generated in this lesson. To do so, just drag and drop the file `library.ddl` from the Explorer window to the Project window (another way: **Product** / **Add text**).

Select this product and execute the command **File** / **Extract** / **SQL**. The DB-MAIN extractor parses the SQL text and produces a physical schema. Apply the reverse engineering script you have developed in Exercise 12.2.

Examine carefully the resulting conceptual schema. Compare it with the original schema (`LIBRARY/Conceptual`). Can you explain the differences?

References

[Batini 1992] Batini C., Ceri S. et Navathe S., B. - *Conceptual Database Design - An Entity-Relationship Approach*, Benjamin/Cummings, 1992.

[Blaha 1998] Blaha M. et Permerlani W. - *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, 1998.

[Date 1999] Date C. J. - *An Introduction to Database Systems*, Addison-Wesley, 1999.

[DBM 1999] *Computer-aided Database Engineering - Volume 1: Database Models*, DB-MAIN Tutorial Series, University of Namur, 1999

[DBM 2002] *DB-MAIN Reference Manual*, University of Namur, 2002

[Elmasri 2000] Elmasri R. et Navathe S. - *Fundamentals of Database Systems*, 3rd Edition, Addison-Wesley, 2000.

[Hainaut 1993] Hainaut, J-L., Chandelon M., Tonneau C., Joris M. 1993a. Contribution to a Theory of Database Reverse Engineering, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, Baltimore, May 1993, IEEE Computer Society Press.

[Hainaut 1994] Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D. 1994. Evolution of database Applications: the DB-MAIN Approach, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

[Hainaut 1996] Hainaut, J-L, Roland, D., Hick J-M., Henrard, J., Englebert, V. 1996. Database Reverse Engineering: from Requirements to CARE tools, *Journal of Automated Software Engineering*, Vol. 3, No. 1 (1996).

[Halpin 1995] Halpin,T., *Conceptual Schema and Relational Database Design*, Prentice-Hall, 1995, ISBN 0-13-355702-2 . Consult also http://www.inconcept.com

[Teorey 1999] Teorey, T., *Database Modeling and Design*, Morgan Kaufman, 1999

Other references from the LIBD on *www.info.fundp.ac.be/libd*

Index

A

access key 3-5, 5-3, 5-10, 12-4
advanced global transformation 11-37
aligning objects 2-11
analysis of
    AUTHOR 7-14
    BOOK 7-5
    BORROWER 8-2
    borrowing 8-7
    closed-borrowing 8-9
    COPY 7-9
    PROJECT 8-7
analysis script 11-28, 11-32
Assist
    Advanced global transformation 11-37
    Global transformation 11-14, 12-16
    Schema analysis 11-29
at least one constraint 6-19
attribute 4-3
    atomic 4-5
    cardinality 4-4
    compound 4-5
    inherited 6-9
    mandatory 4-5
    multivalued 4-5
    optional 4-5
    proper 6-9
    single-valued 4-5
attribute aggregation 6-15

C

cardinality
    of attribute 4-4
    of group 9-35

of role 1-10, 3-3, 4-3
coexistence constraint 6-11
column 5-3
conceptual analysis 7-2
conceptual schema 1-2, 3-2, 4-2, 8-13
copying objects 2-20

D

DBMS 9-2
defining
access key 5-10
attribute 1-8
constraint 4-11, 6-13
entity collection 5-13
entity type 1-7
entity type identifier 1-12
foreign key 5-9
group 4-11, 6-13
project 1-3
rel-type 1-10
rel-type identifier 4-11
schema 1-5
semantic description 1-13
technical description 12-11
disjoint subtypes 6-5
DMS 9-2
downward inheritance 11-26

E

Edit
Copy 2-20
Copy graphic 2-2
Delete 3-2, 3-10
Mark selected 8-21, 11-29
Paste 2-20
entity collection 5-13, 12-8
entity type 4-3

equ 5-8, 9-34, 10-9
equality constraint 5-8, 9-34, 10-9
exactly one constraint 6-19
exclusive constraint 6-17
existence constraint
  at least one 6-19
  coexistence 6-11
  exactly one 6-19
  exclusive 6-17

F

File
  Close project 2-15
  Exit 1-17
  Generate / Standard SQL 3-6, 5-18, 12-11
  New project 1-3
  Open project 2-2
  Print 2-20
  Printer setup 2-20
  Project properties 1-4, 3-2, 4-2
  Report / Textual view 2-18, 3-8
  Save project 1-16
  Save project as 1-16
file 12-8
foreign key 5-3, 5-6, 9-8
  multivalued 9-37

G

generating reports 2-18
global transformation assistant 11-14, 12-16
graphical tool bar 2-2
group
  cardinality 9-35

I

identifier 1-12, 4-3
  hybrid 4-7

multiple 4-6
of rel-type 4-11
primary 4-6
secondary 4-6
index 3-5, 5-3
inheritance 6-2
inverse transformation 6-16
IS-A relation 6-2
IS-A transformation 6-23

L

LIBRARY project
conceptual schema 8-13
logical schema 10-11
physical schema 12-10
SQL code 12-11
logical design 9-1
logical schema 1-14, 3-2, 5-2, 10-11

M

marking objects 8-21, 11-29
move mode 2-10
multiple inheritance 6-5

N

name processing 5-15, 10-12
New
Attribute 1-8
Attribute / First att. 7-15
Collection 5-13, 12-11
Entity type 1-7
Group 4-11, 5-12
Rel-type 1-10
NIAM model 8-20
Note 2-21

O

ORM model 8-20

P

partial subtypes 6-5
partitioned subtypes 6-5
physical design 9-3, 12-1
physical schema 5-17, 12-2, 12-16
predefined script 12-16
prefix access key 5-13, 12-6
primary key 3-5, 5-3
Product
  Copy product 3-4
  New product 1-5
property box
  attribute 1-9
  entity collection 5-14
  entity type 1-7, 6-3
  foreign key 5-11
  group 4-11, 5-10
  identifier 4-11
  project 1-3
  rel-type 1-11
  schema 1-5

Q

quitting DB-MAIN 1-17

R

reference attribute 5-6
reference group 5-6
relational schema 3-5, 5-2, 9-3
rel-type 4-3
  complex 9-13, 9-21
  cyclic 4-13, 8-6
  identifier 4-11

N-ary 4-9, 9-21
with attributes 4-9
reordering attributes and roles 2-16
reverse engineering 5-24, 9-43, 10-19, 12-19
role 1-10, 4-3
inherited 6-9
name 4-13
proper 6-9

S

schema analysis 11-28, 11-29
schema analysis assistant 11-29
schema transformation 6-14, 9-7
script 11-18, 11-28, 11-32
secondary key 5-3
semantic description 1-13
semantics-preserving transformation 6-16, 7-19
shading objects 3-4
SQL code 1-14, 1-15, 3-6, 5-18, 12-11
SQL-compliant 9-3, 11-36
storage space 5-13, 12-8
subtype 6-2
subtype constraint 6-5, 11-4, 11-22
subtype inheritance 6-8
supertype 6-2

T

table 5-3
table identifier 3-5
technical description 12-11
total subtypes 6-5
Transform
Attribute / -> Entity type 7-10, 7-11, 7-15, 7-19, 8-2, 8-16, 10-4, 10-13, 10-14
Attribute / Disaggregation 6-16, 10-3
Entity type / -> Attribute 7-11, 8-18, 11-24
Entity type / -> Rel-type 9-31
Entity type / Add Tech ID 9-18, 9-38, 10-16

Entity type / Rel-types -> is-a 11-23
Group / -> Rel-type 9-11, 9-36
Group / Aggregation 6-15
Name processing 5-15, 10-12
Quick SQL 1-14
Relational model 3-4, 5-3, 12-16
Rel-type / -> Attribute 9-9, 9-31, 10-14
Rel-type / -> Entity type 9-12, 9-27, 10-6
transformation plan 11-6
transformation script 11-18, 11-37
transforming
attributes 6-15, 7-10, 7-19, 8-2, 8-7, 8-16, 10-13
complex rel-types 9-13, 9-21
components of an identifier 7-25
compound attributes 7-23, 10-2, 10-14
cyclic many-to-many rel-types 9-17
cyclic one-to-many rel-types 9-17
entity types 7-11, 8-18, 9-30, 9-38
foreign keys 9-36
identifier attributes 7-23
IS-A relations 11-4, 11-22
many-to-many rel-types 9-11, 9-16, 10-21
multivalued attributes 7-22, 10-4
names 5-15, 10-12
one-to-many rel-types 9-8, 9-15
one-to-one rel-types 9-10
rel-types 9-27, 9-31
single-valued optional attributes 7-20
through scripts 11-18, 11-37

U

upward inheritance 11-23

V

View
Alignment 2-11
Graph. compact 2-4

Graph. standard 2-4
Graphical settings 2-4, 2-5
Text Compact 2-7
Text extended 2-8
Text sorted 2-9
Text Standard 2-7

W

Windows
Graphical tools 2-2

Z

Zoom 2-14