PHENIX Project - BIKIT / FUNDP
IRSIA/IWONL "Tronc Commun" - Contract 5220

PHENIX Project

DATABASE REVERSE ENGINEERING

FINAL REPORT

Volume II : Reverse Engineering

Second Version - April 1993

Volume I : General concepts and Introduction

Volume II : Reverse Engineering

Volume III : Technical appendices

Appendix A : TRANSFORMATION TECHNIQUES FOR DATABASE REVERSE ENGINEERING

Appendix B : PHENIX CARE TOOL - The User's View (Version 2.0)

Appendix C : ADDITIONAL TECHNICAL DOCUMENTS

PHENIX SYSTEM : PHYSICAL DESCRIPTION

PHENIX SYSTEM : SCREEN EXAMPLES

PHENIX SYSTEM : TRANSFORMATIONS SPECIFICATION (FUNCTIONAL)

PHENIX SYSTEM : LIST OF THE TRANSFORMATIONS (TECHNICAL)

PHENIX SYSTEM : OBJECT-BASE CONCEPTUAL SPECIFICATION (Version 2)

PHENIX SYSTEM : NAME PROCESSING (TECHNICAL)

PHENIX SYSTEM : SPECIFICATION OF REAL, THE IMPORT/EXPORT LANGUAGE

PHENIX SYSTEM : SPECIFICATION OF THE INTEGRATION PROCESS

PHENIX SYSTEM : OBJECT-BASE Version 2.03, TECHNICAL SPECIFICATION

PHENIX SYSTEM : OBJECT-BASE OBJECT-BASE IMPLEMENTATION

Chapter 6

NAME PROCESSING

This chapter presents different problems related to naming conventions that are is use is software development. These problems are analysed in the context of database engineering. Some formal rules and heuristics are proposed.

6.1 INTRODUCTION

Name processing is a very important aspect of reverse engineering data bases. This activity will play an important role in many reverse engineering activities.

Name processing covers at least two activities:

- Deciding whether two names are similar. This is the topic of the current chapter.
- Drawing inferences and conclusions from this information. This activity is related to topics such as schema integration, redundancy reduction and so on. We refer to chapter 10 for instance. One possible conclusion drawn from two names being similar is that the objects denoted by these names may be the same.

In the second section we will give an overview of name construction. This section will make the notions of name similarity and name processing clearer.

6.2 NAME CONSTRUCTION

Every data element used in a database and in the programs that process the data will have a name. This name is a string of characters. Data elements are referenced by means of this name.

Although an arbitrary string can be used to refer to a data element we can expect programmers to use 'significant' strings so the meaning of the underlying data element is clear by watching the name. But this is not an absolute rule. Personal preferences and corporate conventions will restrain the names that will be used to denote data elements.

A structure representing a client can be refered to by means of the name CLIENT but CLI, ABCDEF would also do.

Reverse engineering will not be performed on abstract objects but on objects that have become tangible through their name. But names can be constructed in many different ways.

6.2.1 Proper nouns

The name of a data structure is the name used for the object the structure is refering to.

A structure (e.g. a record type) that stands for a client can be named CLIENT

6.2.2 Hierarchy

There may be a hierarchy in the objects and the names of the corresponding data structures may represent this hierarchy. The name will point at the same time to an object and to the position in a hierarchy.

> *The company can make a distinction between individual and corporate clients and the names PRIVATE-CLIENT and CORPORATE-CLIENT will reflect this distinction.*

A multilevel hierarchy can be represented in the same way:

> *A company has employees EMPLOYEE of both sexes - FEMALE-EMPLOYEE and MALE-EMPLOYEE. Depending on their marital status we find SINGLE-FEMALE-EMPLOYEE and MARRIED-FEMALE-EMPLOYEE etc.*

6.2.3 Usage mode and context

Names can be constructed by adding an indication of the use of the name to the proper name of the object the name refers to.

Suppose DATA is the name of a structure ; INPUT-DATA and OUTPUT-DATA show exactly what DATA s used for.

The notion of context is important when we talk about names composed of several parts. One element of the name indicates what object we are talking about. We call this the context of the name. It is the main object. The other parts of the name define the circumstances in which the name appears. In the example above DATA is the context; it's the object we are talking about. INPUT and OUTPUT specify the conditions or the environment in which DATA is used.

6.2.4 Prefix and suffix

A name can be constructed by adding a prefix or a suffix to the context name. The connection can be made by means of a hyphen, an underscore, juxtaposition... There are several possibilities;

- A part may be added in front of the context. The added part is a prefix. A local variable could be named WS-CUSTOMER. The name for the customer is the context and WS can be interpreted as a prefix.
- A string may follow the context. The addition is a suffix.. Say DOSSIER is the name of a data structure and that the application allows a transfer of a dossier. DOSSIER-OLD and DOSSIER-NEW can be used as names.
- A third possibility is the connection of two context names to form a new name. CUSTOMER and INVOICE may be two names. CUSTOMER-INVOICE is a possible name for a relationship between the two objects customer and invoice. In this case it is not clear what part of the name is the context. Both could be the context.

6.2.5 Length - alphabet and reserved words

Varying the length of a name may create other names. The best examples are short names and abbreviations.

The data structure representing a customer can be named CUST by shortening the name CUSTOMER. A number can be named NUM or NR.

Some names will never be used to denote data structures because they have a predefined meaning. In many environments the use of IF, WHILE, DO etc for data structures is not allowed because these names are reserved words in the environments.

6.2.6 Grammatical variants

A name may be created by variations on some properties of a name that exists yet. Such properties are the genus (male and female) and the cardinality (singular and plural).

6.2.7 Corporate rules about names

Some companies impose name formation rules that must be applied for every application. Most of the formation rules mentionned above will be part of such corporate rules (hierarchy, prefix and suffix, using mode ...).

Sometimes not only name formation rules will be imposed but also the names themselves. This practice is useful for large companies whose departments may be spread and whose applications are to be used in many places. The use of corporate imposed names ensures that the same object has the same name in every department. This methodology facilitates integration of several schemas and detection of redundant data. The EMPLOYER entity type for example will always be indicated with the same name : EMPLOYER in each schema, each view, ... on the database.

The names may bear little or no resemblance to the names we would expect the data structures to have.

6.3 NOTION OF NAME SIMILARITY

In the course of reverse engineering, names of data structures will have to be compared to decide whether they denote the same objects and therefor if they are the same or different. These comparisons are important to allow the application of several reverse engineering functionalities.

Similarity is not the same as equality. We may find two different names and still decide that they denote the same object. On the other hand the same name may appear in several contexts and we may conclude that two different objects are denoted.

Similarity means that two names are very much alike in some respects. The resemblance may be syntactical. In that case the names look very much the same. On the other hand the resemblance may be semantical - the names refer to a common meaning.

In the following paragraphs we look at the different 'flavours' of similarity.

6.3.1 Identity

Identity means the names are the same.

NAME in CUSTOMER and NAME in CLIENT are identical.

In the other cases the similarity will not be absolute any more. We will still accept that similar names may denote the same object. The reason to use two or more names for a given object may be explained by program related objectives or by corporate standards.

6.3.2 Prefix and suffix

Two names can be equal except for a prefix or suffix that is added. To decide what part of the name is the prefix or the suffix one should look for the context or the focal concept in the name. In some cases this will be easy like in QTY-04. We see very quickly that QTY is the main concept (quantity). In other cases like in PROD-NUM the context is not so easy to find.

A name and its prefixed or suffixed version can be interpreted as being equal.

> *CUSTOMER and WS-CUSTOMER are similar in that they differ only by the use of the prefix WS.*

But the use of a prefix or a suffix is sometimes an economical way of expressing that two objects are almost alike but not completely. The meaning of the prefix or suffix must be analyzed before deciding on the similarity.

6.3.3 Inclusion

One name is a substring of the other name.

> *e.g CUSTOM and CUSTOMER.*

Care must be taken with a name that is the plural or singular of another name. Other elements in the source must be looked at before deciding that both names refer to the same object.

> *e.g. in the COBOL case an OCCURS clause may reveal that CUSTOMERS is refering to a group of persons rather than to a single person.*

6.3.4 Approximate spelling

Two names are written almost the same.

What evidence is there to decide that both names are similar?

- There may be multiple ways to spell a name:

  e.g. PRODUCT and PRODUKT
- One name may be spelled in a wrong way:

  e.g. CUSTOMMER and CUSTOMER
- The spelling of a name may differ slightly from one language (or dialect) to another:

  e.g. ORGANISATION (French) and ORGANIZATION (U.S. English)
- ...

6.3.5 Language translation

Two names are the same except that they are written in different languages. This is a possibility in a multilingual company where large applications may have been developed by a team of mixed language.

> *e.g. CLIENT, CUSTOMER and KLANT refer to the same object. The names differ only in their linguistic origin.*

6.3.6 Short names or abbreviations

Short names or abbreviations can be used to refer to an object. The meaning of such a short name or abbreviation can often be clear at the spot. Many of them are speaking for themselves.

The distinction between a short name and inclusion is not always clear. The first example is a short name while the third one is a mixture of an inclusion and a short name.

> *e.g. ACCT is used in the meaning of ACCOUNT. NR, NUM are popular short names for NUMBER. CLI is also used in certain fields to denote a CLIENT.*

6.3.7 Synonyms

In this case similarity of names is not any longer a question of lexical similarity or similarity up to a translation. The similarity is rather on the level of the meaning of the name.

> *e.g. SALARY and WAGES can be used both to indicate one's earnings.*

A caveat is necessary here; the use of two names for a single object (earnings are considered as the object in the example above) will often point to a distinction between two 'subtypes' of an object. In the example SALARY could be used for a white collar while WAGES may seem more appropriate for a blue collar worker.

6.3.8 Similarity metrics

By similarity metrics we mean a computable function of the similarity of two or more names.

One possibility to measure the amount of similarity would be to count the percentage of matching characters. This measure can only be applied to the prefix, suffix, inclusion and approximate spelling cases.

In the case of prefixed and suffixed names it is possible to check whether the omission of the prefix or suffix returns a name that is equal to the non-fixed name.

Two problems pop up when dealing with similarity metrics :

- We may be able to compute the similarity of two names but is there a treshold value that distinguishes between similar and dissimilar names?
- Does the same name denote the same object? Two names may be syntactically similar or even equal while the underlying objects may be different. The context in which a name appears plays an important role in taking the decision on similarity or dissimilarity.

6.4 NAME PROCESSING

It was said before that an arbitrary string of characters can be used to refer to an object in most programming languages. When reconstructing a conceptual schema one is dealing with meaningful objects. So it is preferable to have meaningful names refering to the objects.

Processing of names serves several goals :

- Make the names more meaningful to make the manipulation of the underlying objects easier.
- Avoid the presence of homonyms; we don't want names that may have several meanings. As an example take the dutch word PATROON that means at the same time the boss, a bullet and a pattern.
- Avoid the use of synonyms. A single object must be denoted by a single name.

Name processing is essentially a renaming activity. Several modes of processing exist:

6.4.1 Semantic enrichment

There may be corporate conventions and restrictions about the names used in programs. These names may have little meaning.

> *e.g. we may assume that BCLI is a corporate name to denote a client and we can rename the object by CLIENT*

For this kind of semantic enrichment to be possible we must have an idea about the meaning of the corporate names. Documents imposing names for certain objects may be helpful in this regard.

Conformity to an implementation environment may impose other restrictions on names.

CLIENT_NAME may become CLIENT-NAME because the underscore is not allowed in the implementation environment.

6.4.2 Language translation

The language of the application programmer may differ from the language of the reverse engineer. So certain names can be replaced by a translation into the latter's language to ease the manipulation of the names.

6.4.3 Expansion and truncation

We have shown that shortcuts and abbreviations are mechanisms to create names.

To manipulate the underlying objects one may prefer more meaningful names. Shortcut names may be expanded to express the full semantics.

NR can be expanded to NUMBER

Truncation means making names shorter. Names may be too long to manipulate gracefully or a name may function as a tautology.

BUYING-CUSTOMER can be truncated to CUSTOMER because the first part of the name expresses exactly the same as the second part (the example is absurd of course)

6.4.4 Prefixing and suffixing

There are two ways to deal with prefices and suffices.

- A prefix and/or a suffix can be cut off from a name. This is the case when the elimination of it does not influence the semantics of the name.

  WS-CUSTOMER and CUSTOMER express exactly the same if we may assume that WS points to a local variable

- A prefix or suffix can be added to a name to add semantics. This may be the case if a corporate rule is not followed throughout the program. It can also be used to remove synonyms.

  NAME may appear several times e.g. as the attribute of a record type PRODUCT and of an attribute type CUSTOMER. To resolve the ambiguity of NAME we can add a prefix to NAME so we have two clearly distinct names PRODUCT-NAME and CUSTOMER-NAME.

One should be very careful with the removal of prefices and suffices because they can be very important in reconstructing a conceptual schema. The fact that a prefix or suffix is added may indicate that the meaning of the underlying object may differ from one context to another. It can also be an indication of subtypes etc.

The presence of the same prefices and/or suffices in several names can be used as an indication that these names belong semantically together. As such we may have an indication of an entity type (see later on).

So care must be taken before touching the prefix and/or suffix. The exact meaning of its use should be clear before manipulating it.

6.4.5 String replacement

In the extreme case this comes down to replacing the object's name with a completely new string of characters that is lexically unrelated to the original name. This kind of replacement is possible when the original name doesn't refer to the meaning of the underlying object at all.

All the other techniques mentionned like prefix/suffix manipulation, truncation, expansion, translation and so on are 'mild' forms of string replacement. Only part of the name is replaced. And replacement should be taken in a very broad sense. It may also mean 'eliminated'.

6.5 RENAMING OBJECTS

The processing of names is a renaming manipulation. A new name is given that is closer to the meaning of the object the name is refering to. Some problems arise when renaming objects so the scope of renaming must be defined very clearly.

The notion of the scope is important here. The scope is that part of the name set that is influenced by the renaming activity.

6.5.1 Local renaming

The scope of a local renaming is the name of a single data element. The presence of the original name in another context can lead to a problem since there is no longer an identity similarity between the original and the new name. A closer analysis of this second name is necessary to determine the similarity with the renamed object.

6.5.2 Global renaming

The scope in the case of global renaming may vary greatly. The scope may be as wide as the set of all the names in the program. The scope may be limited to all the record types in a logical file or all the attributes in a single entity type. Global renaming will always be applied to a set of names.

It is important to specify the scope of renaming.

Some techniques are fit for a program wide renaming like the elimination of technical names.

Some will be limited to the attributes of an entity type like the removal of a meaningless prefix or suffix.

Chapter 7

SCHEMA TRANSFORMATIONS

This chapter defines the concept of schema transformation as a basic technique for database engineering, particularly useful in reverse engineering. Possible applications of each transformation are suggested.

7.1 INTRODUCTION

After a short intuitive definition of the concept of transformation, we will state the objectives of schema transformations, their possible uses, and the framework we will use in the following sections to describe each of them.

7.1.1 Short definition

Many processes applied in data structure processing, such as database forward or reverse engineering, can be characterized as transformations.

Transforming a schema consists in applying only structural modifications to this schema, keeping (as much as possible) its semantics unchanged. An accepted way to define the term "semantics" is the whole set of real-world facts which can be represented by the schema. So semantics-preserving schema transformations do not modify the possible real-world type of facts that can be represented by the schema on which they are applied.

Example (figure 7.1) : schema S_2 is the result of transformation of schema S_1. Any fact which can be represented in S_1 (an employee, his number and address; a contract, its

number and date; a link between an employee and his contracts) can also be represented in S_2, and conversely.

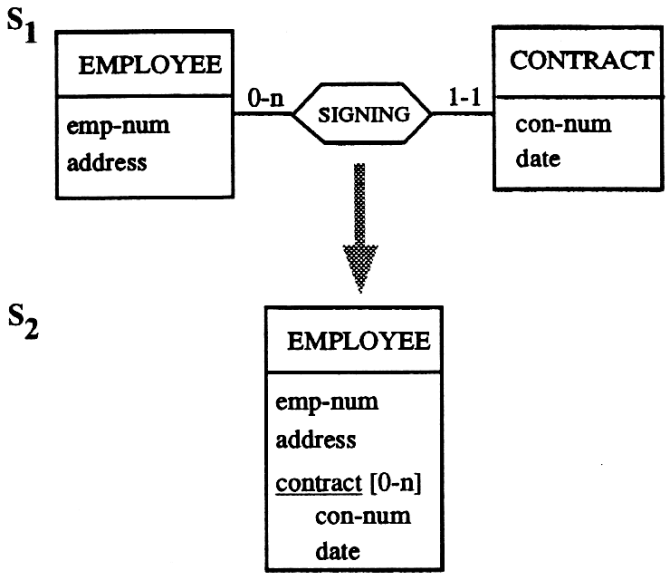


Figure 7.1 *- Example of transformation*

Though a more formal presentation of the concept of schema transformation can be useful, this short illustration will be sufficient for the purpose of this manual. This chapter will describe some of the most useful schema transformations as far as database reverse engineering is concerned. The technical annex will include a more extensive catalog of techniques.

7.1.2 Objectives of transformations

Let us recall a summarized view of the forward and reverse engineering processes, as far as the schema level is concerned (see chapters 3 and 5 for a more general explanation) :

- the **conceptual design** process is the first phase of forward engineering. It consists, among others, in improving a conceptual schema in order to make it closer to the conceptual user's view. Data structures normalization can also be placed in this step.
- the **logical/physical design** goal is to produce from a conceptual schema an optimized DBMS-compliant implementation of it, decreasing in this way its expressiveness.
- the **reverse engineering conceptualization** recovers a conceptual schema from a physical one, and particularly the original expressiveness of the schema. This process is therefore very similar to the conceptual design step. Sometimes, we will call them both simply "conceptualization".

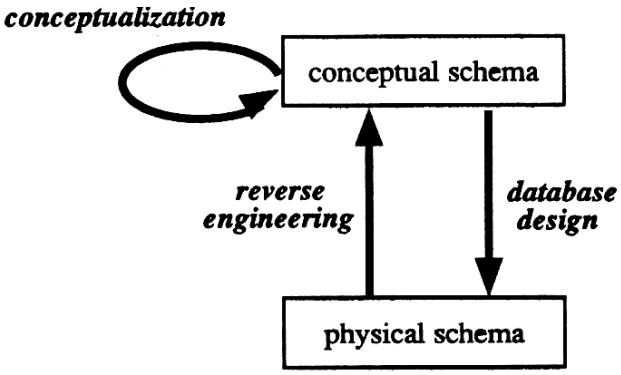


Figure 7.2 *- Three processes where schema transformations are useful*

Schema transformations are basic techniques to achieve the objectives of these three processes :

- in **physical/logical design**, transformations allow obtaining a physical optimized solution which incloudes all the semantics present in the conceptual schema, but expressed in structures which are understandable by a D(B)MS.
- in **reverse engineering conceptualization**, transformations help to decode physical structures and re-expressed them in a more (conceptual) understandable way. Since the objective is to know the contents of the physical schema, the extracted structures must be kept unchanged, at least during a first analysis.

  Very often, reverse transformations of those used in physical database design have to be used during this "decoding" work. Therefore, a good knowledge of these database design transformations helps detect in the physical structures whether they have been applied. Thanks to this detection, reverse transformations can be usefully applied.
- In **conceptual design** too, the semantics-preserving characteristics of the transformations allow the user to progress incrementally to his most expressive "ideal" schema, being dispensed of semantics equivalence checkings.

Qualities of more conceptual structures are developed in chapter 14, section 14.4, but we want to mention here *conciseness* (several transformations are concerned with "merging") and *clarity* (structures very close to the mental representation of informations, see for instance the promoting/demoting transformations).

7.1.3 Composition of transformations - Global transformations

The transformation presented hereafter are basic operations that can be performed on data structures. They are intended to cover most useful data structure restructuring techniques. However, they are very basic and their preconditions are sometimes perceived as too restrictive, according to the aim of the user. The solution to bypass these constraints consists in using macro-transformations composed of a sequence of basic transformations. That allows a greater variety of behaviors, such as, among others :

- a transformation works only on one object. To apply the transformation on n (n>1) objects, iterate it n-1 times, using the intermediary results.
- the transformation in mind does not apply on a construct to be processed. Use first a transformation which replaces this construct by a more basic one, then execute the transformation in mind.

  Example : The transformation of a relationship type into reference attributes does not allow multidomain roles in this relationship type. The solution consists in using first the transformation of a multidomain role into several relationship types, then applying the first transformation.
- a transformation works only on one object. To apply the transformation on n objects, use first a merging transformation.

  Example : It is not possible to transform several attributes into a single entity type. When the intention is to split the set of attributes of an object, use first the transformation which aggregates several attributes into a single one. Then it is possible to apply the latter transformation.
- a transformation works only on one kind of objects. To apply the transformation on other kinds of objects, promote or demote them.

Another more knowledgeable use of transformations consists in applying the same transformation to a set of objects, for instance of a whole schema, instead of only one. This allows to spare repetitive individual transformations.

7.1.4 Framework of transformation description

The transformations presented in the next sections are described as follows (the hurried reader could skip the precise definition; but for using the transformation, all parts are necessary) :

- **Principles** : gives a short and general description of the transformation
- A real world **Example**
- **Goals** : objectives and results of the transformation are discussed, using the distinction defined in 7.1.2.

  Note : each reversible transformation can be used for one particular practical goal : the undo of its reverse transformation. This particular use will not be recalled later.

- **Triggering situations** : when is the transformation to be used? Typical situations which it could be useful to apply the transformation to, are described. It is important to distinguish the preconditions of a transformation, which must be fulfilled to apply the transformation, from these triggering situations, which are additional hints or suggestions. Note that the preconditions, when complex, are often the first obvious triggering situation.
- **Precise definition** : the transformation is given a general *graphical illustration*, precise *preconditions* and *actions*.

Transformations presented in this manual are the most important ones. Other (less important) transformations will be developed in the technical annex. In particular, the concept of generalization/specialization is not managed, neither in the basic transformations, nor in transformations that are specific to this concept.

Transformations are grouped in the following pages according to the kind of objects they can be applied to. However, considering the number of transformations, the following table will give a more global view of them. For each transformation we give its name, its main class of application, and its reverse transformation.

Transformation	Main use	Reverse transformation
transforming an ET into a RT	conceptualization	transforming a RT into an ET
splitting an ET	conceptualization	merging two ET
transforming an ET into an attr.	conceptualization	transforming an attr. into an ET
merging two ET's	conceptualization	splitting an ET
transforming a RT into ref. attr.	physical database design	transforming ref. attr. into a RT
transforming a RT into an ET	conceptualization	transforming an ET into a RT
splitting a RT with a multidomain role into several ones	conceptualization (splitting)	merging similar RT into a multidomain role
merging similar RT into a multidomain role	conceptualization (merging)	splitting a RT with a multidomain role into several RT.
transforming an attr. into an ET	conceptualization	transforming an ET into an attr.
replace a compound attr. by its components	physical design	aggregate a list of attr. into a compound attr.
transforming a multivalued. attr. into a list of attr.	physical design	transforming a list of similar attr. into a multivalued attr.
transforming ref. attr. into a RT	reverse engineering	transforming a RT into ref. attr.
aggregate a list of attr. into a compound attr.	reverse engineering	replace a compound attr. by its components
transforming a list of similar attr. into a multivalued attr.	reverse engineering	transforming a multivalued. attr. into a list of attr.

7.2 TRANSFORMING AN ENTITY TYPE

7.2.1 Transforming an entity type into a relationship type

Principles

An entity type which is related with other entity types (at least two) by 1-N relationship types only is transformed into a relationship type.

Example

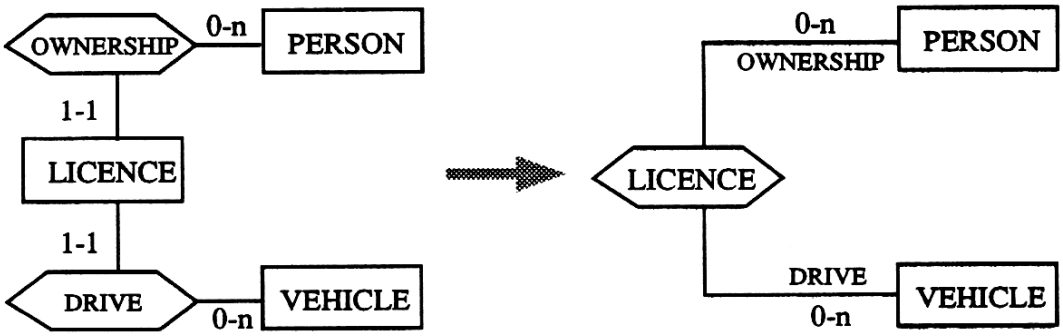


Figure 7.3 *- Example of transformation of an entity type into a relationship type*

Goals

This transformation is mainly reverse engineering-oriented. It is totally neutral from a semantical point of view. So its goals are limited :

- in **reverse engineering conceptualization**, this transformation will allow to redefine a relationship type implemented by a record type, when the target DBMS does not provide, for instance, N-N or cyclic relationship types.
- in **conceptual design**, it can be used to present an entity type and its environment (i.e. its related entity types) more concisely, and/or to present it more as a link between other entity types than as an independent object.

Triggering situations

- An entity type has few attributes (date is often a typical attribute of a link)
- An entity type has only two or three 1-N relationships.
- An entity type has a name which is the concatenation of some entity types which it is linked to.

- The name of an entity type is a verb
- An entity type has not a local identifier, made up of own attributes only. Some components of its identifier are roles.
- The semantics of real-world objects described by an entity type is closer to links than to independent objects.

Definition

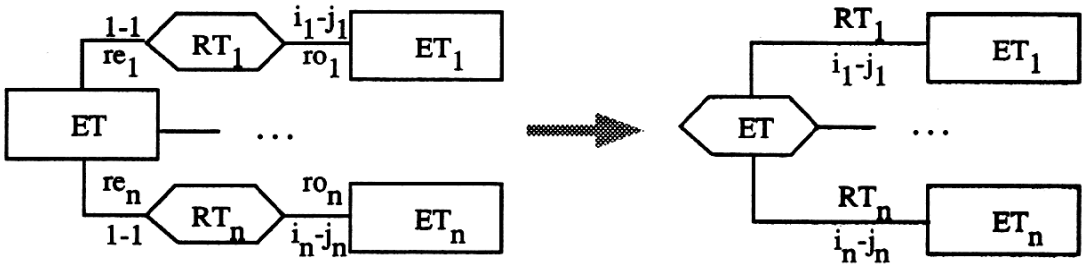


Figure 7.4 *- Transformation of an entity type into a relationship type*

Preconditions

- ET must have at least two relationship types.
- All the relationship types RT_i are 1-N (more precisely the cardinalities of re_i must be 1-1), binary, non-cyclic, without attributes.

  Note : ro_i can be multidomain

Actions

1. Replace the entity type ET by a relationship type with the same name, the same attributes and the same identifiers/keys
2. Each relationship type RT_i of ET is replaced by a role with :

   a) the name of RT_i

   b) the same cardinalities as ro_i

   c) the same identifier participations as ro_i

7.2.2 Splitting an entity type

Principles

The attributes and roles of an entity type are distributed into two new entity types replacing the first one. These two entity types are linked by a 1-1 relationship type.

Example

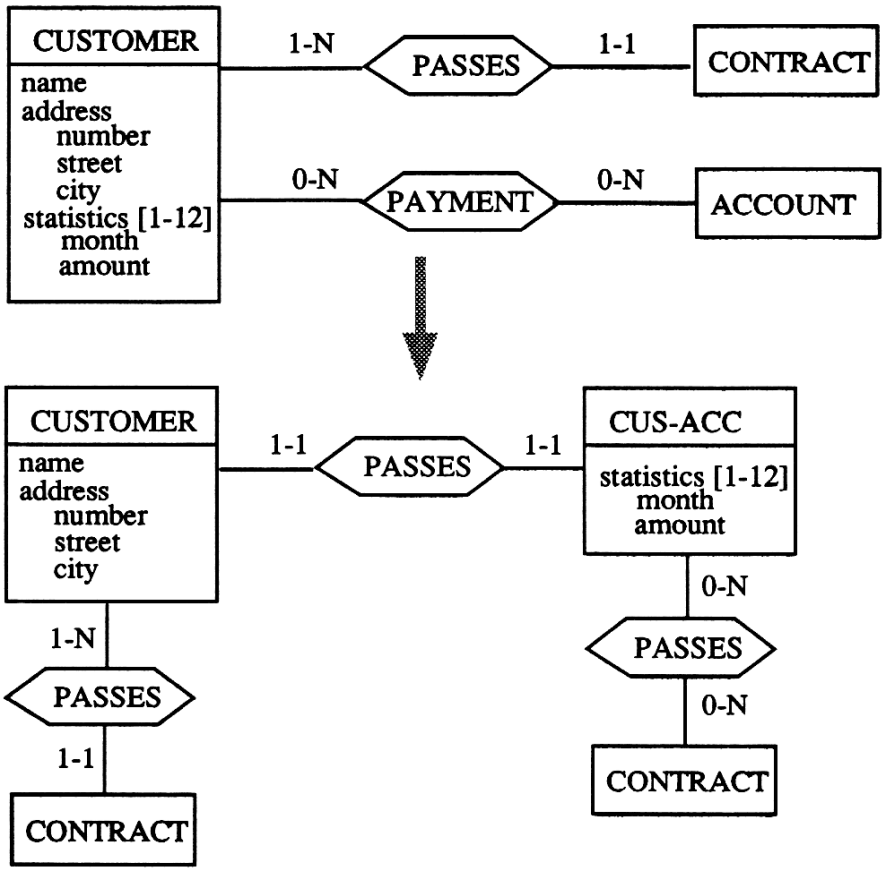


Figure 7.5 *- Example of splitting an entity type*

Goals

- In **physical/logical design**, this transformation is usually used for physical purposes. By splitting a record type, it is possible to define different access or storage strategies for each resulting record types.
- In **conceptual design**, this transformation can be used for concept refinement, that means the disaggregation of a too global concept.
- In **reverse engineering**, particularly for file management systems, when record types are usually aggregated, this transformation is one possible way to split record types.

Note : this transformation has no links with the normalization theory : ***it cannot be used to normalize an entity type***. In order to normalize an entity type by decomposition, the transformation of an attribute into an entity type has to be used, possibly preceded by an aggregation of the attributes.

Triggering situations

- An entity type is large. It has many attributes.
- The semantics of an entity type is not precise. It contains several subconcepts linked by a bijection.
- There are existence dependencies between attributes and/or roles.

Note : we will not describe here how the selection of attributes/roles to be exported is decided (by user, by some more automatic rules, ...).

Definition

see Figure 7.6

Preconditions

ET must have at least two attributes/roles.

Actions

1. create a second entity type ET-2
2. create a 1-1 relationship type between ET and ET-2
3. transfer the user-selected attributes and/or roles on ET-2
4. for each identifier/key of ET :

   if it was composed only of transferred attributes and/or roles, then these attributes/roles identify (are a key of) now ET-2. Otherwise, they still remain identifiers of ET.

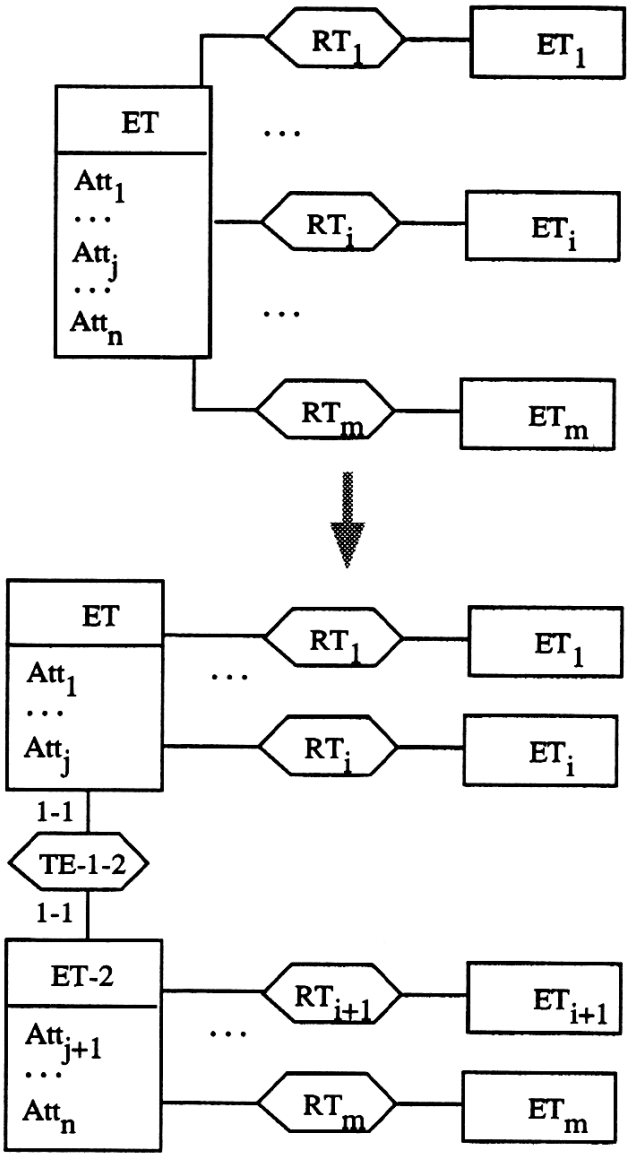


Figure 7.6 *- Splitting of an entity type into several ones*

7.2.3 Transforming an entity type into an attribute

Principles

An entity type which is related with one and only one entity type is transformed into an attribute of this entity type.

Example

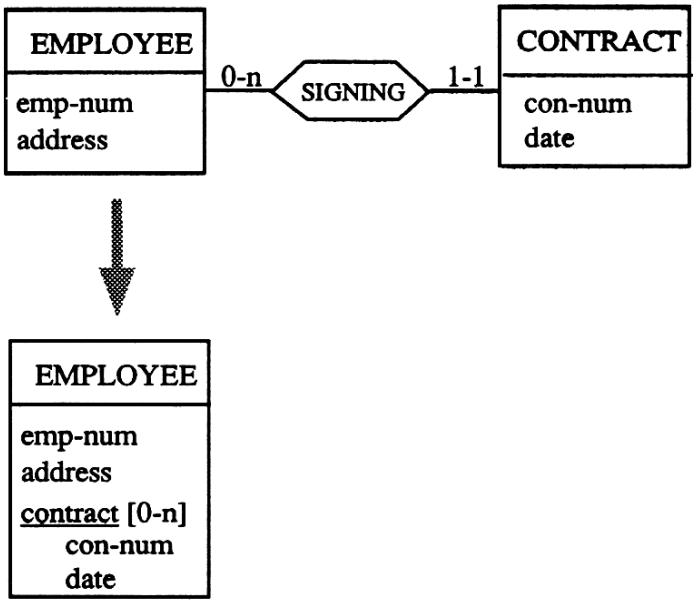


Figure 7.7 *- Example of transformation of an entity type into an attribute*

Goals

- in **physical design**, this transformation is used :

  -to delete a relationship type, when the target DBMS does not authorize them.

  -to improve access between the two entity types

  -for storage purposes

- in **reverse engineering** conceptualization, it can be used to aggregate too much divided record types, as usually in RDBMS, for instance.
- in **conceptual design**, to reduce the importance of an entity type, which becomes a subconcept of another one.

Note : the resulting entity type can be less normalized than the previous structure

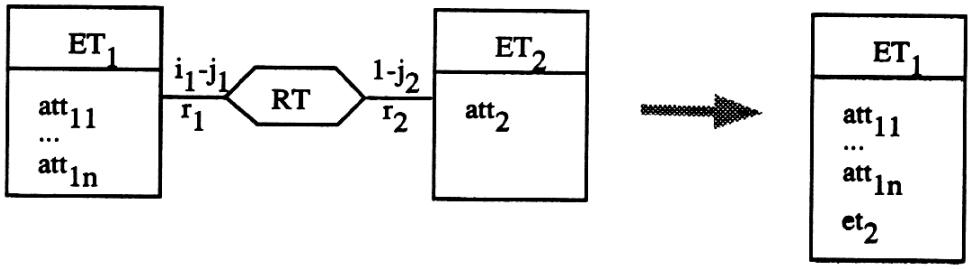
Triggering situations

- see preconditions.
- An entity type has only one attribute (whose semantics is close to VALUE), which could identify it.

- An entity type has in one of its identifiers, a role of another one.

Definition

CASE 1 (ET_2 has only one attribute)



CASE 2 (ET_2 has several attributes)

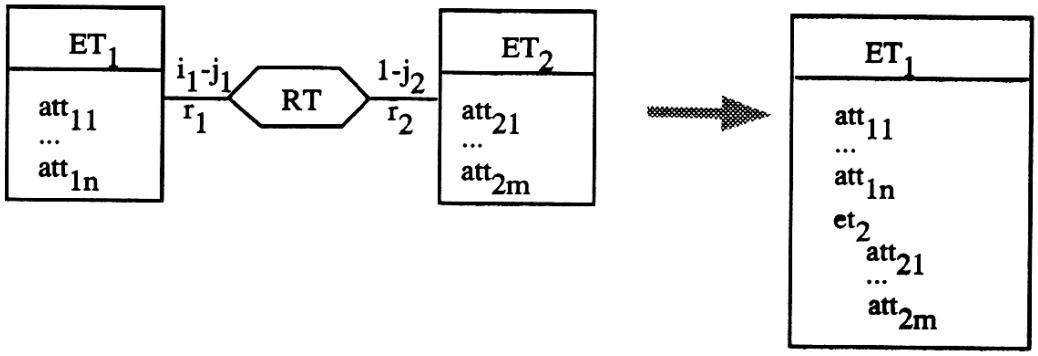


Figure 7.8 *- Transformation of an entity type into an attribute*

Preconditions

- ET_1 and ET_2 are linked by one and only one binary relationship type RT, without attributes, with single-entity roles. The minimal cardinality of the role of ET_2 in RT must be 1.
- ET_2 has at least one attribute
- ET_2 has at least one identifier. All its identifiers are made up of local attributes of ET_2 only, and possibly r_1.
- In case 1 (ET_2 has only one attribute), either the cardinality of r_1 is 1-1, or the repeating factors of att_2 is 1-1.

Actions

1. create an attribute et_2 of ET_1, with the same name as ET_2.
2. transfer the identifier/key participations of r_2 to et_2.
3. ***CASE 1*** : ET_2 has only one attribute att_2

   a) $rep(et_2) = card(r_1) * rep(att_2)$

   b) if card (r_2) = 1-1 and r_1 does not belong to an identifier of ET_2 with att_2, et_2 is an identifier of ET_1.

   c) transfer the possible attributes of att_2 to et_2.

   CASE 2 : ET_2 has several attributes $\{att_{21}, ..., att_{2m}\}$

   d) transfer $\{att_{21}, ..., att_{2m}\}$ to et_2.

   e) $rep(et_2) = card(r_1)$

   f) if card (r_2) = 1-1,

      i. the identifiers of ET_2 made up of a strict subset of $\{att_{21}, ..., att_{2m}\}$ identify now ET_1

      ii. if $\{att_{21}, ..., att_{2m}\}$ was an identifier of ET_2, then et_2 is an identifier of ET_1.
4. delete RT
5. delete ET_2

7.3 TRANSFORMING A SET OF ENTITY TYPES

7.3.1 Merging entity types

Principles

Two entity types linked by a 1-1 relationship type are merged into a single one, which inherits all the attributes and roles of these two entity types.

Example

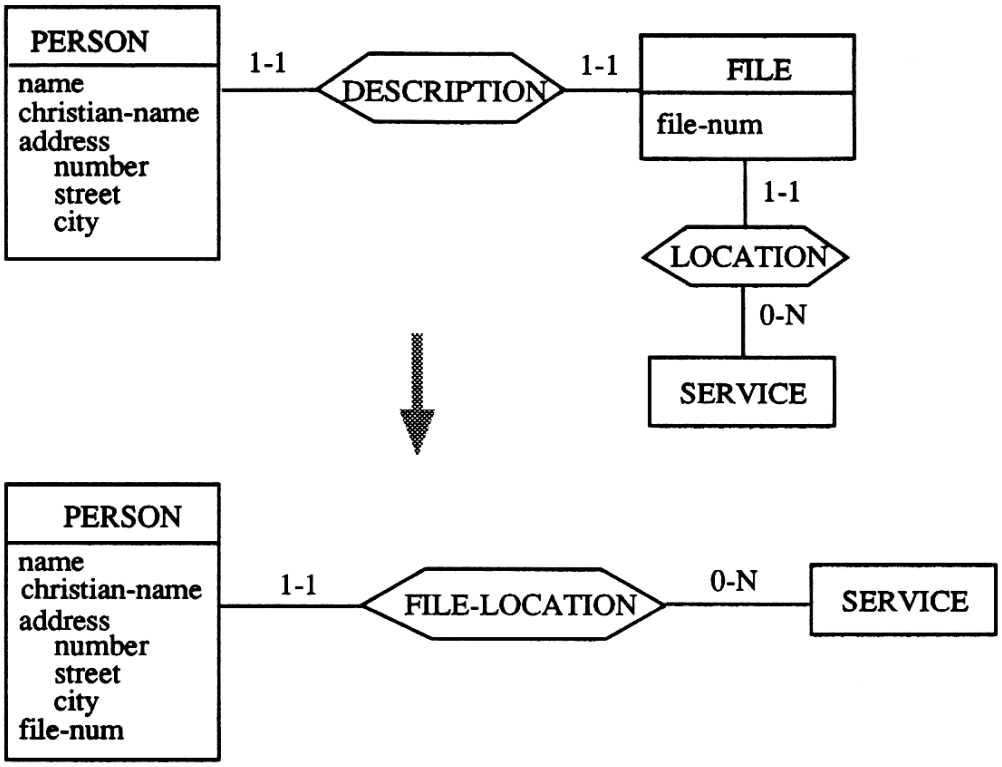


Figure 7.9 *- Example of merging of two entity types into a single one*

Goals

- In **physical/logical design**, this transformation is usually used for physical purposes. By merging two record types, it is possible to improve storage and access between the merged record types.
- In **conceptual design**, this transformation can be used for concept abstraction, that means the aggregation of two concepts into a global one.
- In **reverse engineering** conceptualization, particularly for relational systems, when record types are usually split, this transformation is one possible way to merge record types.

Note : this transformation has no links with the normalization theory. The result of this transformation is as normalized as the original entity types.

Triggering situations

Two entity types, linked by a bijection :

- small entity types (few attributes)
- which could be aggregated to form a structure which represents a real-world fact type.

Definition

see Figure 7.10

Preconditions

- two different entity types ET-1 and ET-2
- linked only by one and only one 1-1 relationship type, without attribute

Actions

1. transfer all the attributes/roles of ET-2 (or ET-1) to ET-1 (ET-2), with their identifier, key or constraint participations.
2. transfer the identifier, key or constraints of ET-2 (ET-1) to ET-1 (ET-2).

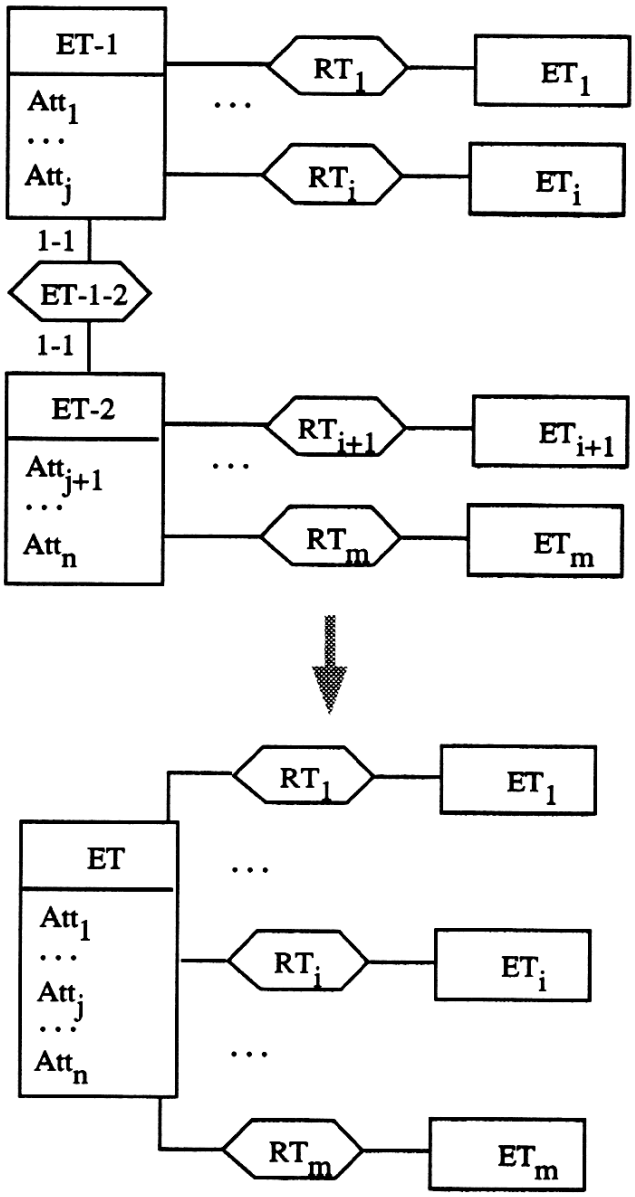


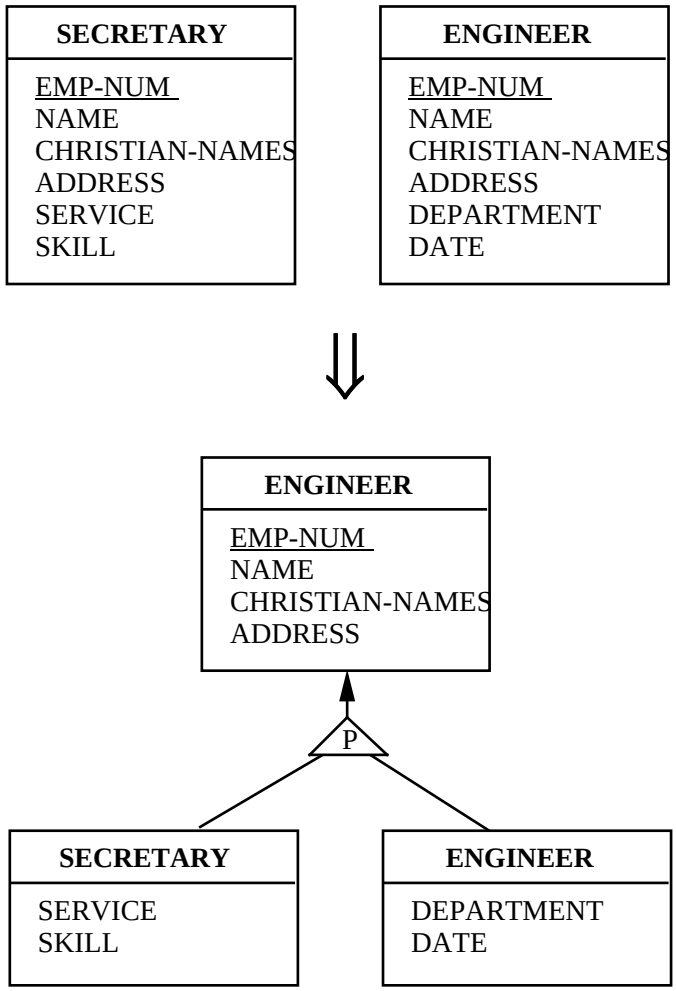
Figure 7.10 *- Merging two entity types into a single one*

7.3.2 Making an entity supertype with entity subtypes.

Principles

Sebveral entity types have attributes and/or roles that seem to be similar. Their are given a common supertype that collects these similar components.

Example



Goals

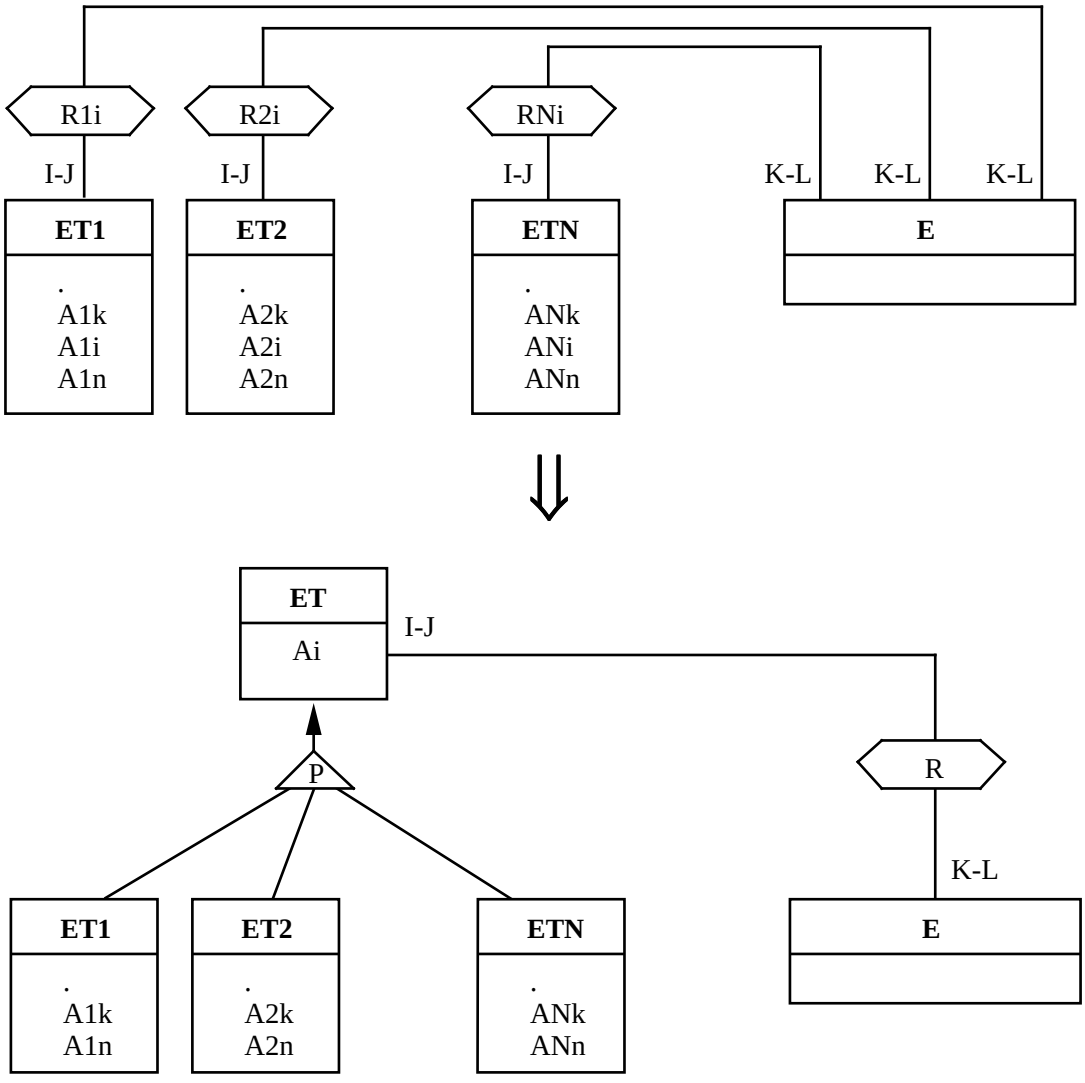
This transformation will be used mainly in conceptual normalization.

- in conceptual design, to make subtype/supertype structures explicit during the step-by-step building of conceptual schemas;
- in conceptual design, to normalized a conceptual schema;
- in reverse engineering, to clean a basic conceptual schema.

Triggering situations

- there exist several entity types that represent classes of similar objects;
- these have similar attributes or appear in similar roles; by *similar* we mean that they have identical, or compatible domains, and a close meaning.

Definition



Preconditions

Entity types ET_1, ET_2, .., ET_N (N≥2) are characterized by one or two of the following conditions :

- similar attributes : E_1has attribute A_{1i}, E_2 has attribute A_{2i}, .., E_N has attribute A_{Ni}; A_{1i}, A_{2i}, .., A_{Ni} have the same domain; more than one set of such attributes may exist;
- similar roles of similar rel-types : E_1 appears in rel-type R1i, E_2 appears in rel-type R2i, .., E_N appears in rel-type RNi; their cardinalities are the same (I-J); the other

entity type(s) of the rel-types are identical (E), and have the same cardinalities (K-L); more than one set of such rel-types may exist;

- A global identifier is defined on E_1, E_2, .., E_N; the respective components of this identifier are similar components;

Actions

1. create entity type ET
2. for each set of similar attributes $\{A_{1i}, A_{2i}.., A_{Ni}\}$, do :

   - add an attribute A_i to ET; A_i has the domain of A_{1i};

   - delete attribute $\{A_{1i}, A_{2i}.., A_{Ni}\}$;
3. for each set of similar rel-types $\{R_{1i}, R_{2i}.., R_{Ni}\}$, do :

   - create rel-type R between ET and E, with cardinalities I-J and K-L;

   - delete rel-types $\{R_{1i}, R_{2i}.., R_{Ni}\}$;
4. for each global identifier :

   - define an identifier on ET with the components correspondant to those of the global identifier;

   - delete this global identifier.

7.4 TRANSFORMING A IS-A HIERARCHY

see the three techniques in the technical annex

7.5 TRANSFORMING A RELATIONSHIP TYPE

7.5.1 Transforming a relationship type into reference attributes

Principles

A 1-N relationship type between two entity types is replaced by a reference, made up of attributes, in the entity type on the N-side to the entity type on the 1-side.

Example

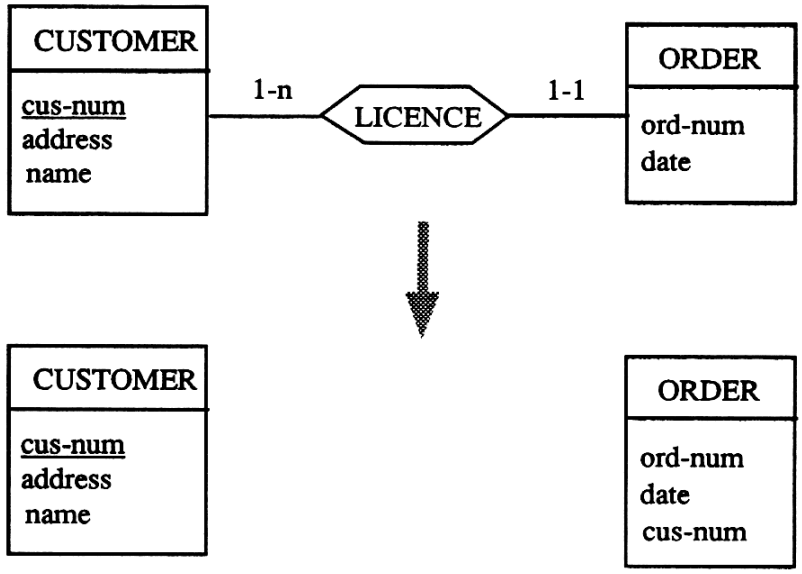


Figure 7.11 *- Example of transformation of a relationship type into reference attributes*

Goals

This transformation is typically a **physical design** transformation. It is used to eliminate the concept of relationship type in a whole schema, when the target D(B)MS does not provide it (Cobol, RDBMS for instance)..

Triggering situations

see preconditions.

Definition

Note : we do not take into account here the access transformations, i.e. how access paths are translated into access keys.

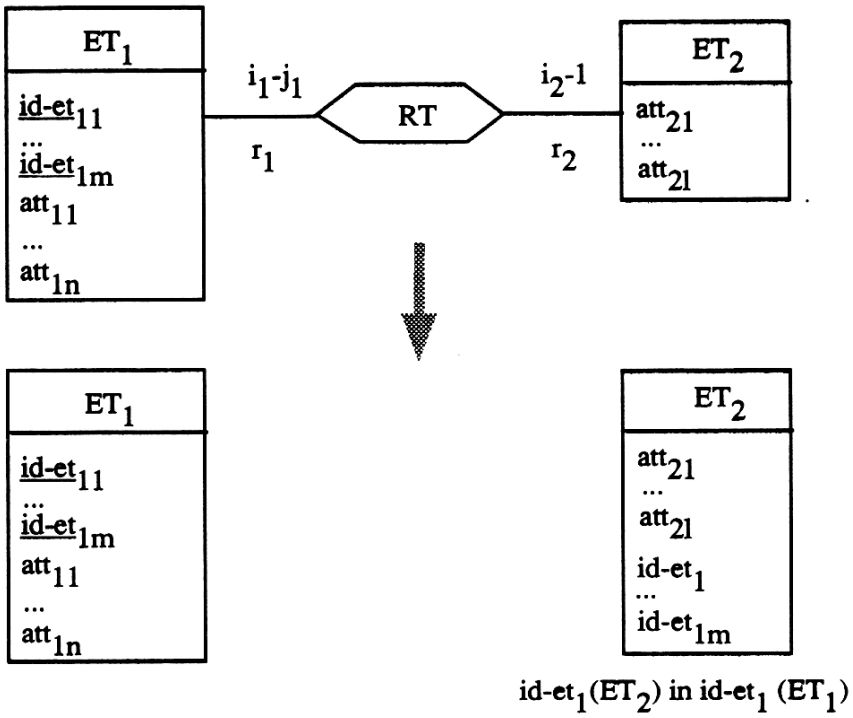


Figure 7.12 *- Transformation of a relationship type into reference attributes*

Preconditions

- RT is binary, without attributes
- r_1 and r_2 are monodomain
- max-card(r_2) is 1
- ET_1 has an identifier only made up of attributes (id-et_{11}, ..., id-et_{1m})

Actions

1. create for each attribute id-et_{1k} (1_k_m) a copy in ET_2.

   a) min-rep(id-et_{1k}(ET_2)) = i_2

   b) min-rep(id-et_{1k}(ET_2)) = 1

   c) create an inclusion constraint from id-et_{1k}(ET_2) to id-et_{1k}(ET_1).

   d) if i_1 is 1, create an inclusion integrity constraint from id-et_{1k}(ET_1) to id-et_{1k}(ET_2).
2. replace the identifier participations of $r_1$by id-et_{11}, ..., id-et_{1m}.
3. if j_1 is 1, id-et_{11}, ..., id-et_{1m} is an identifier for ET_2.
4. delete RT

7.5.2 Transforming a relationship type into an entity type

Principles

A relationship type is transformed into an entity type, its roles being transformed into relationship types.

Example

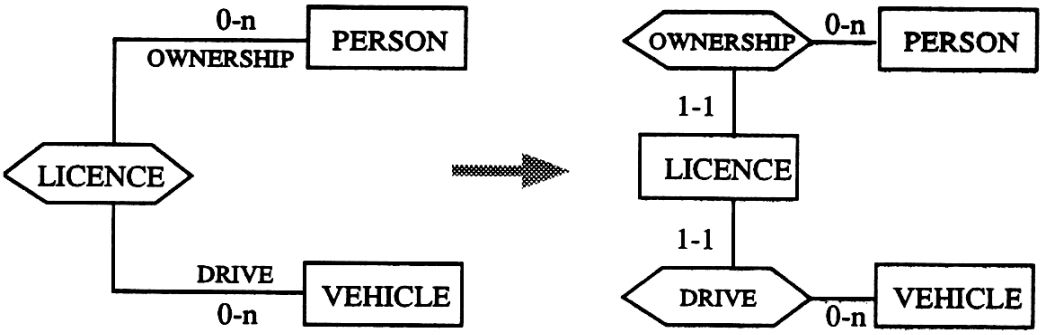


Figure 7.13 *- Example of transformation of a relationship type into an entity type*

Goals

- In **physical design**, this transformation is used to obtain data structures without
  - N-N relationship types
  - relationship types with attributes

  It is often used at the beginning of the design, because few D(B)MS support these two concepts.
- In **conceptual design**, this transformation allows the promotion of a relationship type into a more important and independent concept, which new relationship types could be defined on.

Triggering situations

- A relationship type has many attributes.
- Real-world objects described by a relationship type can be perceived more adequately as independent objects than as links.
- A relationship type has an identifier made up of own attributes and few roles.
- A new relationship type must be defined on a relationship type.
- A role must be considered as a relationship type, for instance to add it some attributes.

Definition

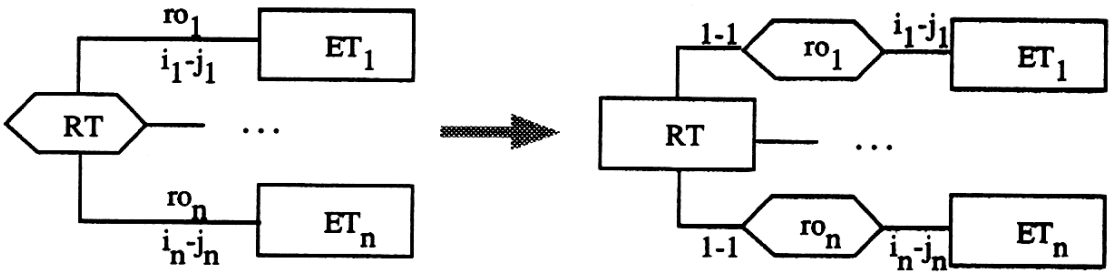


Figure 7.14 *- Transformation of a relationship type into an entity type*

Preconditions

No preconditions for this transformation.

Actions

1. Replace the relationship type RT by an entity type with the same name and the same attributes.
2. Each role ro_i (which can be multidomain) of RT is replaced by a binary relationship type with :

   a) the name of ro_i

   b) a role 1-1 on RT side

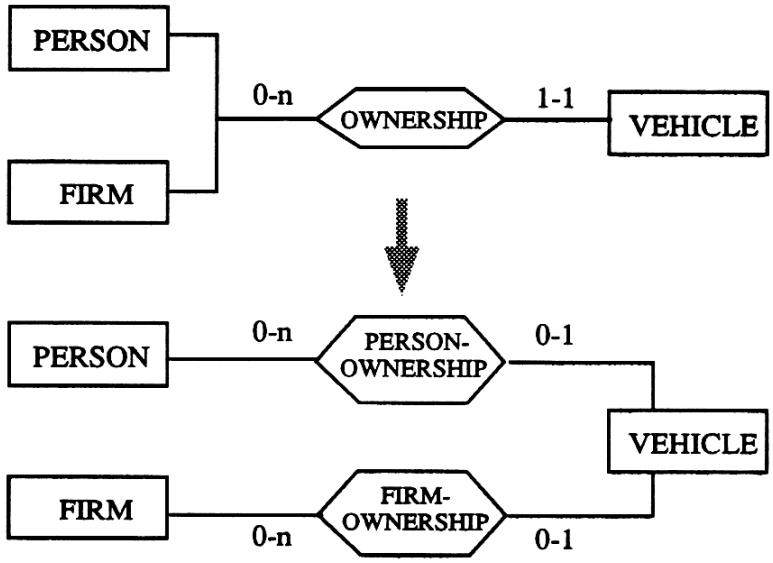
   c) a role on the other side with the same cardinality as ro_i, the same participating entity type(s) as ro_i, the same identifier/key participations as ro_i.

7.5.3 Splitting a relationship type with a multidomain role into several ones

Principles

A relationship type containing a multidomain role is split into several similar relationship types, one for each entity type participating in this role.

Example




```
excl : PERSON-OWNERSHIP.VEHICLE,FIRM-OWNERSHIP.VEHICLE
```

Figure 7.15 *- Example of splitting a relationship type with a multidomain role into several ones*

Goals

This transformation is a degradation transformation : the resulting schema is less concise; a constraint has been produced.

- In **physical design**, this transformation is used to eliminate multidomain roles, because few D(B)MS support this concept.
- In **conceptual design**, this transformation is in fact a specialization process, creating a disjunction of specializations of the original relationship type. This process can be perceived as a degradation, which can be useful if these specializations are significant, for instance to change role cardinality or add new attributes to the resulting relationship types.

Triggering situations

Multidomain roles are often used at a high level of conceptualization. A relationship type with a multidomain role could be transformed into several relationship types when :

- it has optional attributes, whose existence depends upon the entity type in the multidomain role.
- it has wide cardinalities, restricted by constraints according to the entity type in the multidomain role.
- it has too large semantics. It represents too many different kinds of objects (links), according to the entity type in the multidomain role.

Definition

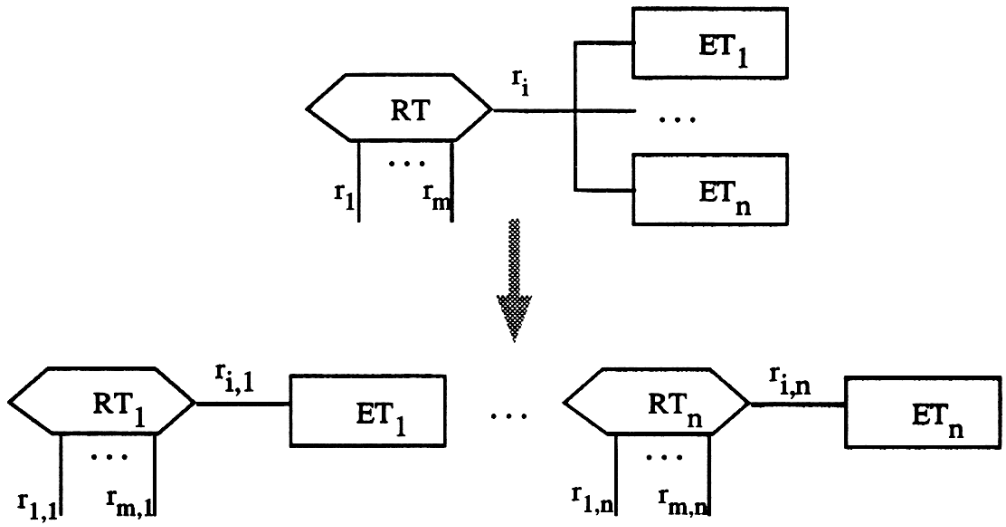


Figure 7.16 *- Splitting a relationship type with a multidomain role into several ones*

Preconditions

No preconditions for this transformation.

Actions

1. For each entity type ET_j ($1 \leq j \leq n$) of the role r_i, create a relationship type RT_j :

   a) whose name is similar to RT

   b) with the same roles r_k ($k \neq i$) (same participating entity types - same name - same maximal cardinality but the minimal one is set to 0).

   c) with a role r_i only played by ET_j (same name - same cardinalities).

2. For each entity type ET_j ($1 \leq j \leq n$) of the role r_i, create the constraint :

" e ET_j : min-card(r_i) ≤ number of RT_1, ..., RT_n of e ≤ max-card(r_i)"

3. transfer the identifier participations of each r_k ($k \neq i$) to one of $\{r_{k,1}, ..., r_{k,n}\}$, according to the identified object.
4. transfer the identifier participations of r_i to $(r_{i,1}, ..., r_{i,n})$
5. delete RT

7.6 TRANSFORMING A SET OF RELATIONSHIP TYPES

7.6.1 Merging similar relationship types into a multidomain role

Principles

Several relationship types which are very similar, except the entity type(s) participating to a role, are merged (generalized) into a single one.

Example

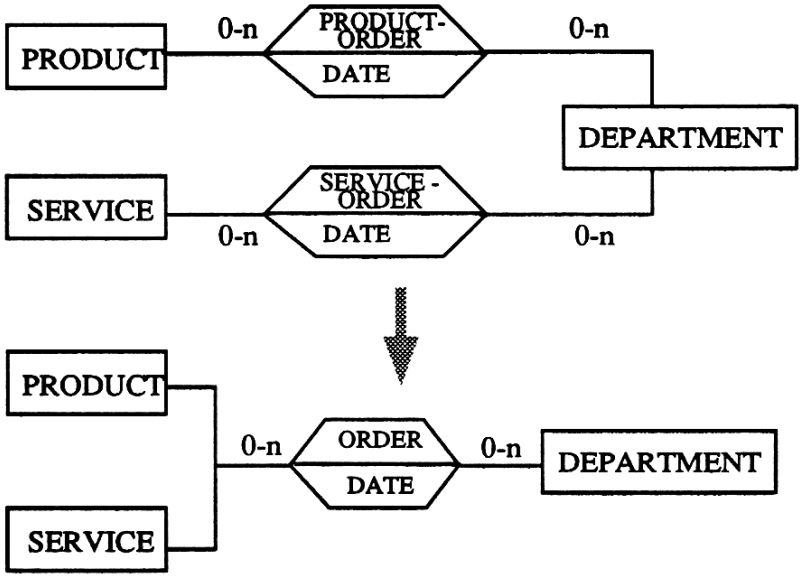


Figure 7.17 *- Example of merging of similar relationship types into a multidomain role*

Goals

This transformation is a **conceptualization** transformation. It implements the generalization process, replacing several relationship types by a more general one, which describes the union of the original relationship types. Its goals therefore are also the generalization ones :

- to obtain a much more concise structure (let n be the number of merged relationship types; n-1 relationship types are in fact removed)
- to obtain a more general relationship type which stresses the similarities of the original relationships.
- In counterpart, some specific informations (cardinalities, names, existence dependencies) are less explicit. They must be of small importance.

Triggering situations

This transformation will be used on relationship types whose semantics are close. Hints of this situation are based on similarity :

- similar names. Typical names are composed of a common part and a specific part corresponding to the name of the specific entity type (see example).
- similar attributes (except few ones)
- similar roles (name - cardinalities - participating entity types), except one which has different participating entity types.
- there exists a global cardinality constraints for similar roles.

Definition

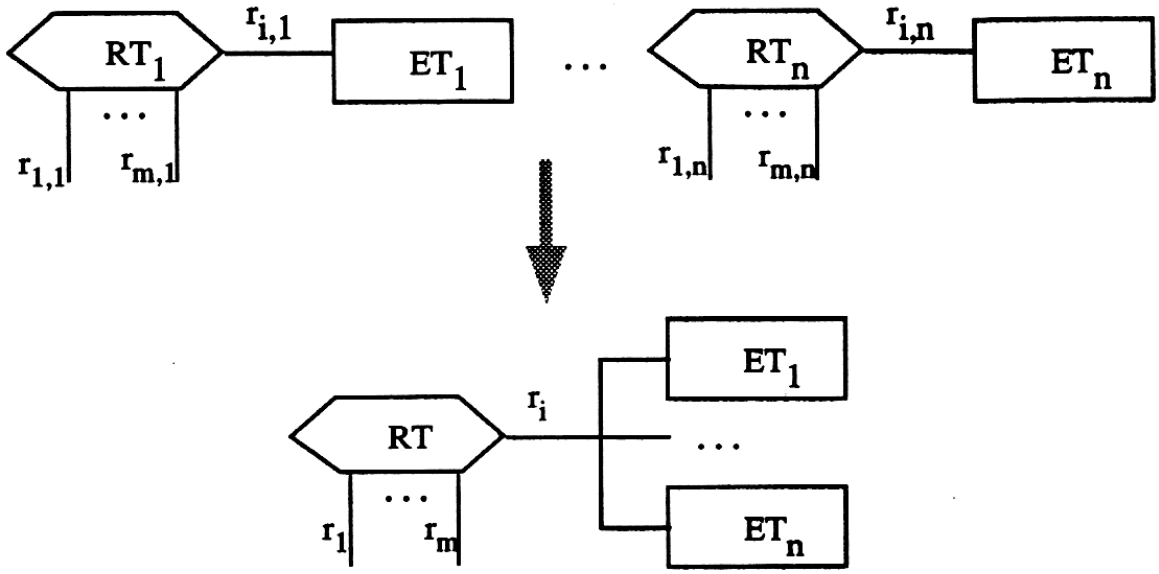
Note : the following definition could be generalized to include the generalization of relationship types. Because some theoretical work has still to be done, it is not yet presented hereafter.

see Figure 7.18

Preconditions

Let RT_1, ..., RT_n n ($n \geq 2$) relationship types such that :

- all relationship types have the same degree (m)
- The n roles $r_{i,k}$ ($1 \leq k \leq n$) are monodomain and involves different entity types (however, they have the same semantics).
- $\forall$ j ($1 \leq j \leq m$; $j \neq i$) : the n roles $r_{j,k}$ ($1 \leq k \leq n$) are similar, that means they have at least the same participating entity type(s).



∀ j (1≤ j≤ m; j≠i) : the n roles rj,k (1≤ k≤ n) have the same entity type(s)

Figure 7.18 *- Merging of similar relationship types into a multidomain role*

Actions

1. create a relationship type RT with :

   a) the role(s) r_j (1≤ j≤ m; j≠i) with, for each r_j, the same participating entity types as the similar roles $r_{j,1}$, ..., $r_{j,n}$.

   i.∀ r (1≤ j≤ m; j≠i) : min-card (r_j) = Σ_k min-card ($r_{j,k}$) (1≤ k≤ n)

   (except if there is a constraint of global minimal cardinality defined on some $r_{j,k}$)

   creation of a constraint of specific minimal cardinality for each original role $r_{j,1}$, ..., $r_{j,n}$

   ii ∀ r_j (1≤ j≤ m; j≠i) : max-card (r_j) = Σ_k max-card ($r_{j,k}$) (1≤ k≤ n)

   (except if there is a constraint of global maximal cardinality defined on some $r_{j,k}$)

   creation of a constraint of specific maximal cardinality for each original role $r_{j,1}$, ..., $r_{j,n}$

   b) the role r_i with all the participating entity types of the roles $r_{i,1}$, ..., $r_{i,n}$.

   i. min-card (r_i) = min {min-card ($r_{i,k}$); 1≤ k≤ n}

   ii. max-card (r_i) = max {max-card (r_i); 1≤ k≤ n}

   (if needed, creation of a constraint of specific cardinalities for each original role $r_{i,1}$, ..., $r_{i,n}$)

2. delete RT_1, ..., RT_n.

7.7 TRANSFORMING AN ATTRIBUTE

7.7.1 Transforming an attribute into an entity type

Principles

An attribute of an entity type is transformed into an entity type, which is related by a new relationship type to its original father.

Example

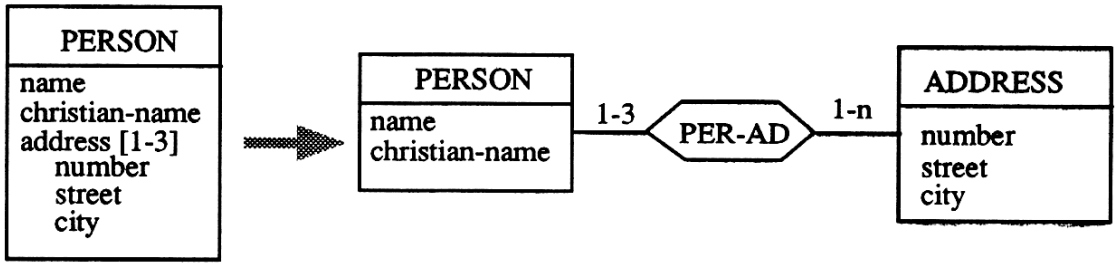


Figure 7.19 *- Example of transformation of an attribute into an entity type*

Goals

- in **physical design**, this transformation can be used

  -to define different access or storage strategies for each resulting entity type.

  -to eliminate multivalued or compound attribute, if the target D(B)MS does not manage them

- in **reverse engineering** conceptualization, it can be used to split too much aggregated record types (frequent in file management systems, such as COBOL).
- in **conceptual design**, the attribute is promoted into a more important and independent object, on which new relationship types could be defined.
- This transformation is an implementation of the decomposability theorem in the relational theory. It allows the **normalization** of the schema in some cases : the functional dependencies between subattributes of the transformed attribute are moved from the entity type to the new one.

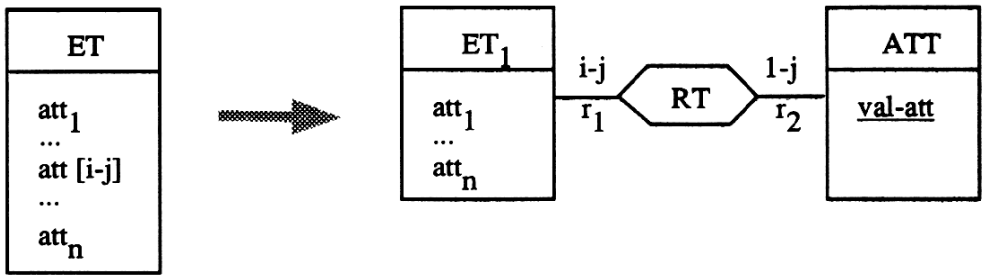
Triggering situations

- an attribute is made up of many subattributes.
- the semantics of an attribute is closer to object description than to value recording.

- an attribute is identified by some of its subattributes.
- an attribute contains subattributes which functional/existence dependencies are defined on.
- a new relationship type must be defined on an attribute.
- an attribute is multivalued.

Definition

CASE 1 (atomic attribute)



CASE 2 (compound attribute)

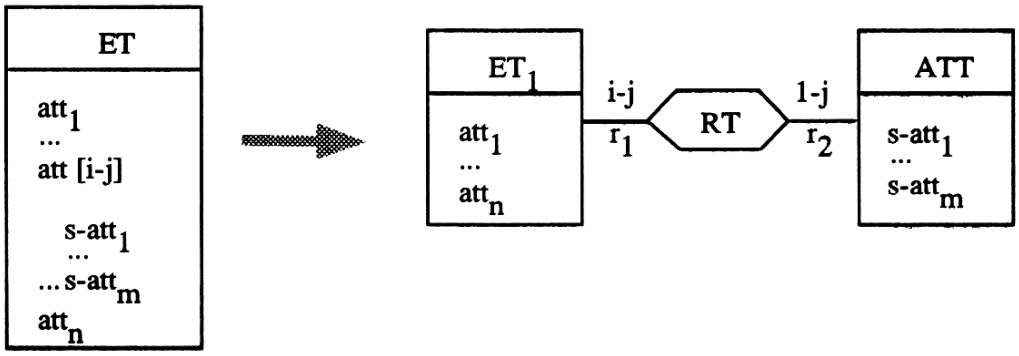


Figure 7.20 *- Transformation of an attribute into an entity type*

Two variants will be presented. The first one has wider preconditions than the second one. That means that the two variants can be applied to the situations which fullfill the preconditions of the second one.

FIRST VARIANT

This variant creates a new entity type whose each entity represents a value of att. The semantics of this new entity type is *value-oriented* in that each ET entity represents one distinct value of the attribute.

Preconditions

- att is a direct (i.e. first level) attribute of ET

  Note : this precondition can be released by a recursive application of this transformation.

Actions

1. create an entity type ATT, with the same name and the same identifier(s) as att.
2. **CASE 1** : att is atomic

   create an attribute val-att, whose repeating factors are 1-1 and which identifies ATT.

   CASE 2 : att is compound of s-att$_1$, ..., s-att$_m$

   transfer s-att$_1$, ..., s-att$_m$ to ATT (cardinalities unchanged).
3. create a relationship type RT, with a name "name(ET)-name(att)".

   a) card(r_1) = rep(att)

   b) min-card(r_2) = 1

   c) if att or (in CASE 2) a subset of s-att$_1$, ..., s-att$_m$ identify(ies) ET,

   max-card (r_2) = 1 else max-card (r_2) = n

   d) for each identifier/key of ET, if it is made up only of some s-att$_k$, then these attributes identify (are a key of) now ATT.

   e) The identifier participations of att are replaced by r_2.
4. **CASE 2** : if ATT has no identifier, create an identifier made up of s-att$_1$, ..., s-att$_m$.
5. delete att

SECOND VARIANT

This variant creates a new entity type whose each entity represents a particular instance of a value of att to describe an entity of ET. This variant is *instance-oriented* in that the each ET entity represents one instance of the attribute (several instances of the same value may exist).

Preconditions

- att is a direct (that means of first level) attribute of ET

  Note : this precondition can be released by a recursive application of this transformation.
- att is multivalued
- att is not an identifier of ET

Actions

1. create an entity type ATT, with the same name and the same identifier(s) as att.
2. **CASE 1** : att is elementary

   create an attribute val-att, whose repeating factors are 1-1 and which identifies ATT, with the role r_1 defined hereafter.

   CASE 2 : att is compound = {s-att$_1$, ..., s-att$_m$}

   transfer s-att$_1$, ..., s-att$_m$ to ATT (cardinalities unchanged).
3. create a relationship type RT, with a name "name(ET)-name(att)".

   a) card(r_1) = rep(att)

   b) min-card(r_2) = 1

   c) max-card (r_2) = 1

   d) for each identifier/key of ET, if it is made up only of some s-att$_k$, then these attributes identify (are a key of) now ATT, with the role r_1.

   e) The identifier participations of att are replaced by r_2.
4. **CASE 2** : if ATT has no identifier, create an identifier made up of s-att$_1$, ..., s-att$_m$, with the role r_1.
5. delete att

7.7.2 Replace a compound attribute by its components

Principles

A compound attribute is replaced by its component attributes, which are therefore redefined one level higher. The aggregation link between these attributes is lost.

Example

see Figure 7.21

Goals

This transformation is a **degradation** transformation. The aggregation of the component attributes is lost. This loss can be interesting in the following situations :

- in **physical design**, to eliminate all the compound attributes, for instance when it is required by the target DBMS (relational, for instance).

  Note : the first normal form of the relational theory imposes this elimination.

- in **reverse engineering** conceptualization, to eliminate a (non-semantical) artificial compound attribute, for instance created during the design for physical reasons.

  Example : in COBOL, artificial compound fields are often created because record keys can be defined on one field only.

- in **conceptual design**, when the compound attribute has lost its importance.

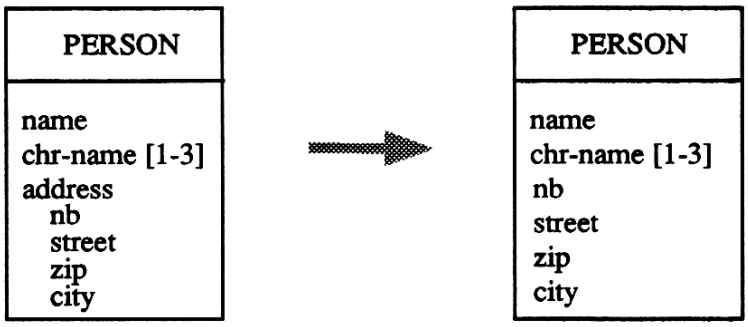


Figure 7.21 *- Example of replacement of a compound attribute by its components*

Triggering situations

- a compound attribute has no semantical name.
- a compound attribute has 1-1 repeating factors and its components too.

Definition

Note :The father of an attribute can be either an entity type, either a relationship type or an attribute. This genericity is expressed by a special graphical representation, which will be used herebelow and several times in this chapter.

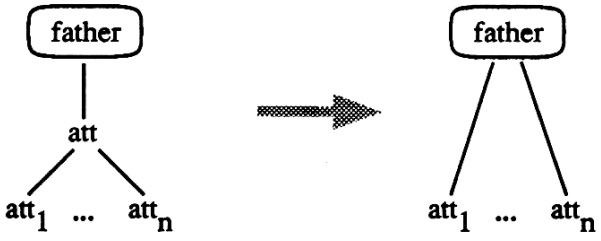


Figure 7.22 *- Replacement of a compound attribute by its components*

Preconditions

- att must be a compound attribute
- no identifier is defined on att
- rep(att) = 1-1 or $\forall$ i : rep(att_i) = 1-1 ($1 \leq i \leq n$)

Actions :

1. change the parent of att_1, ..., att_n to object type *father*
2. $\forall$ i $(1 \leq i \leq n)$: rep(att_i) is set to rep(att) * rep(att_i)
3. replace the identifier, key and constraint participation of att by att_1, ..., att_n.
4. delete att

7.7.3 Transforming a multivalued attribute into a list of attributes

Principles

A multivalued attribute is replaced by a list of similar simple attributes.

Example

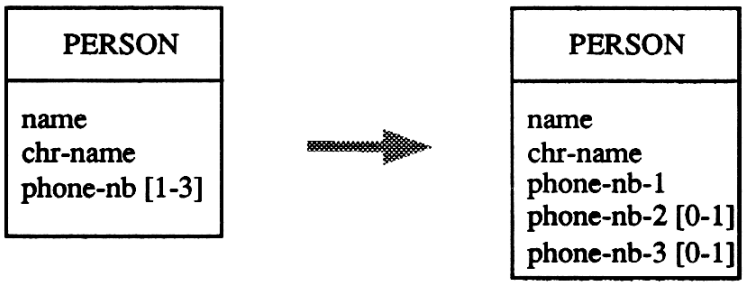


Figure 7.23 - *Example of replacement of a multivalued attribute by a list of attributes*

Goals

This transformation can be considered mainly as a **degradation** transformation. The equality of domain is better expressed in the original structure than in the resulting one. This loss could be interesting in the following situations :

- in **physical design**, to eliminate all the multivalued attributes, for instance when it is required by the target DBMS (standard relational, for instance).

  Note : the first normal form of the relational theory imposes this elimination.

- in **reverse engineering** conceptualization, to eliminate a (non-semantical) artificial multivalued attribute, created for instance during the design for physical reasons.
- in **conceptual design**, when the multivalued attribute is perceived as less important than each individual sets of values, for instance to rename them.

Triggering situations

- a multivalued attribute has no semantical name.
- the minimal and the maximal repeating factors of a multivalued attribute are equal.

Definition

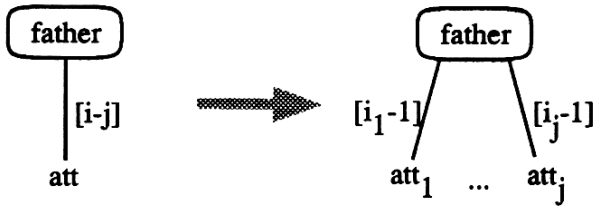


Figure 7.24 *- Replacement of a multivalued attribute by a list of attributes*

Preconditions

- att must be a multivalued attribute : j _ 2
- no identifier is defined on att.
- att, and any of its components, does not participate in any identifier or key.

Actions

1. create j attributes att_1, ..., att_j for the father of att
2. $\forall$ k (1≤ k≤ j) :

   a) name(att_k) is, for instance, name(att) suffixed by k.

   b) max-rep(att_k) is 1.

   c) if k ≤ i, then min-rep(att_k) is 1 else min-rep(att_k) is 0.

   d) if att is compound, make a copy of all its components for att_k.
3. delete att

7.8 TRANSFORMING A SET OF ATTRIBUTES

7.8.1 Transforming reference attributes into a relationship type

Principles

One or several attributes of an entity type, which references an entity type (there is an inclusion constraint from these attributes to the identifier) are replaced by a relationship type linking these two entity types.

Example

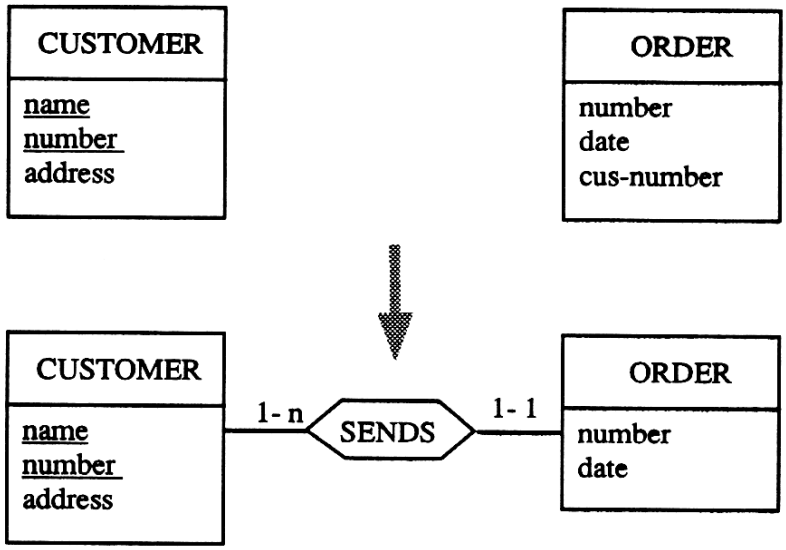


Figure 7.25 *- Example of transformation of reference attribute(s) into a relationship type*

Goals

This transformation is a typical reverse-engineering oriented transformation. The reference attribute(s) record a semantical link. This transformation allows to better express this link with the best concept in the E/R model to do it, i.e. the relationship type. This improvement is usable in two main situations :

- in **reverse engineering** conceptualization, to re-create semantical relationship types, converted during the design when the target DBMS doesn't provide the concept of relationship type (R-DMBS, Cobol Files, ...)
- in **conceptual design**, when the user has first perceived only a weak link, materialized by reference attribute(s), whereas it should be better to express a relationship type (this confusion is frequent, see [Storey90]).

Triggering situations

The situations presented hereafter are helpful to detect a referential integrity constraint :

- the name(s) of one or several attributes of an entity type contains the (short) name of an entity type.
- the name(s) of one or several attributes of an entity type contains hints of its (their) referential nature (for instance, the substring "ref").
- the name(s) of one or several attributes of an entity type is similar to the name of one or several attributes forming a reduced identifier of an entity type.
- there exists an explicit constraint, or some verifications of it in the program, typically in updating processes, that each value of one or several attribute(s) of an entity type must exist also for one or several attributes forming an identifier of an entity type.

Definition

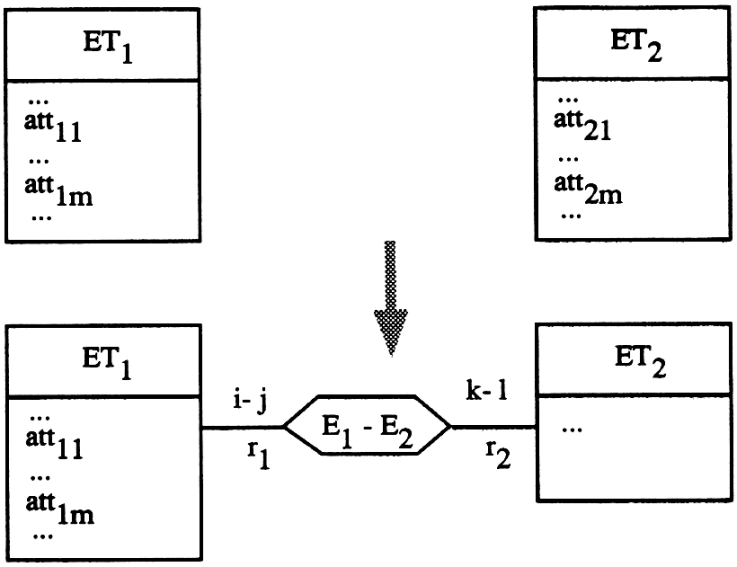


Figure 7.26 *- Transformation of reference attribute(s) into a relationship type*

Preconditions

- att_{11}, ..., att_{1m} is an identifier of E_1.
- the length of the concatenation of att_{11}, ..., att_{1m} is equal to the length of the concatenation of att_{21}, ..., att_{2m}.
- att_{21}, ..., att_{2m} must have the same repeating factors.
- there is an inclusion constraint : att_{21}, ..., att_{2m} is-in att_{11}, ..., att_{1m}.

Actions

1. create a relationship type, whose name could be the concatenation of E_1 and E_2.
2. min-card(r_1) = 0 (it can be 1, if an equality constraint can be asserted).
3. max-card(r_1) = 1 if att_{21}, ..., att_{2m} is an identifier of E.

   max-card(r_1) = n if not.
4. max-card(r_2) = min-rep(att_{21}).

   max-card(r_2) = max-rep(att_{21}).
5. if att_{11}, ..., att_{1m} is a key of E_1, then the access $ET_2 \rightarrow ET_1$ is provided in the new relationship type.

   if att_{21}, ..., att_{2m} is a key of E_2, then the access $ET_1 \rightarrow ET_2$ is provided in the new relationship type.
6. delete att_{21}, ..., att_{2m}.

7.8.2 Aggregation of a list of attributes into a compound father attribute

Principles

A list of attributes are aggregated into a new compound attribute. They are therefore redefined one level lower.

Example

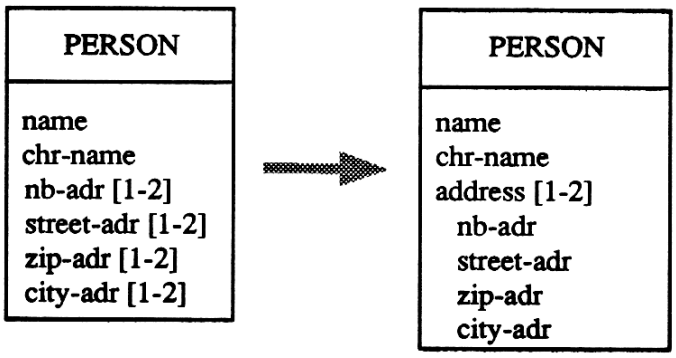


Figure 7.27 *- Example of adding a new father attribute to a list of attributes*

Goals

This transformation is an enrichment transformation. It creates a new aggregation link between the attributes, materialized by an intermediary attribute. This transformation is useful :

- in **physical design**, when the target DBMS accepts some constructs (constraints, keys, instructions, ...) only for one attribute (and not for several ones). Artificial compound attributes are therefore created in this case.
- in **reverse engineering** conceptualization, to recover compound attributes, which were eliminated because the DBMS does not admit this concept.
- in **conceptual design**, when a list of attributes are better perceived as members of a meaningful aggregation, than as individual separated concepts.

Triggering situations

- some (and not all) (generally contiguous) attributes of a same father have a common substring in their name
- some (and not all) (generally contiguous) attributes of a same father have the same repeating factors (except 1-1).

Definition

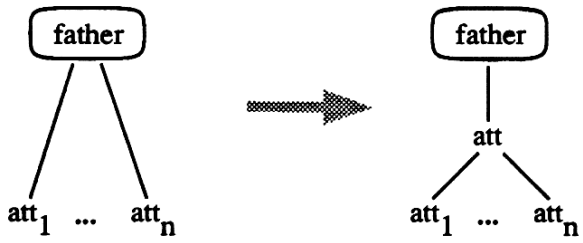


Figure 7.28 *- Adding a new father attribute to a list of attributes*

Preconditions

- att_1, ..., att_n are n ($n \geq 2$) attributes of a same father

Actions

1. create an attribute att of father, with a user-defined name, and with repeating factors 1-1.
2. change the father of att_1, ..., att_n to att.
3. if att_1, ..., att_n have the same repeating factors i-j different from 1-1, and with user agreement, then

   a) $\forall$ k ($1 \leq k \leq n$) : $rep(att_k) = 1\text{-}1$

   b) rep(att) = i-j

4. replace att_1, ..., att_n by att in all the identifiers/keys, they are altogether a member of.

7.8.3 Transforming a list of similar attributes into a single multivalued attribute

Principles

A list of similar attributes are replaced by a single multivalued attribute, which groups all the values of these attributes.

Example

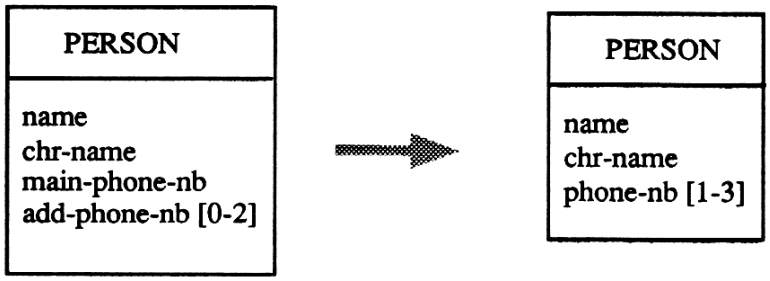


Figure 7.29 *- Example of transformation of a list of similar attributes into a single multivalued attribute*

Goals

This transformation is mainly an enrichment transformation. It allows to better express the similarity of the attributes in the resulting structure. This transformation can be used :

- in **reverse engineering** conceptualization, to recover semantical multivalued attributes, implemented by a list of attributes because the target DBMS does not accept multivalued fields.
- in **conceptual design**, when the similarity of the attributes is so strong that their grouping into a multivalued attribute is perceived as more important than their individual existence (otherwise, this transformation can be perceived as a degradation transformation).

Triggering situations

- several (but not all) (contiguous) attributes with the same domain have a common substring in their name. Typically, the name differences consists only of a numbering.
- several (but not all) (contiguous) attributes with the same domain have the same repeating factors (except 1-1).

Definition

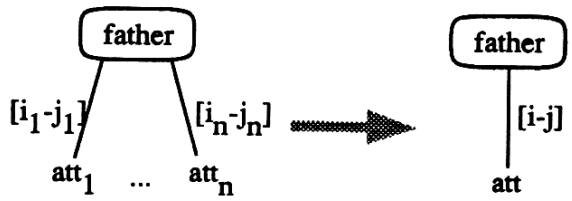


Figure 7.30 *- Transformation of a list of similar attributes into a single multivalued attribute*

Preconditions

- att_1, ..., att_n are n (n _ 2) simple attributes of a same father.
- att_1, ..., att_n have the same domain (and particularly the same length).

Actions

1. create an attribute att of father, whose name could be a common substring of the names of att_1, ..., att_n.

   The domain of att is the domain of att_1, ..., att_n.

2. min-rep (att) = Σ_k min-rep(att_k) ($1 \leq k \leq n$).

   max-rep (att) = Σ_k max-rep(att_k) ($1 \leq k \leq n$).

3. if each att_k ($1 \leq k \leq n$) identifies (are a key of) the same object, then att now identifies (is a key of) this object.

4. delete att_1, ..., att_n.

Chapter 8

REPRESENTATION PROBLEMS IN DATABASES

This chapter analyses the various ways data can be used for representing real-world facts, and the problems that may occur. Data are organized according to two levels, namely data instances and data types. Data instances are to represent real-world facts while data types generally represents real-world fact types. Problems occur when facts and fact types are represented more than once. This chapter is an introduction to the following two chapters, dedicated to Data redundancy and Multiple views.

8.1 INTRODUCTION

The definition of the database as a representation of a part of the real world is the very basis of the so-called database approach to application design, a design approach that originated in the seventies. We have shown that this paradigm is still valid in the context of the reverse engineering of old applications (chapter 5).

This chapter provides a deeper insight on the relationships that hold between databases and the real world. Understanding these relationships is necessary to classify the problems that will be encoutered in the reverse engineering processes. In particular, they make an adequate framework for studying the data redundancy and the multiple-view problems.

In order to avoid too many backward references to chapter 2, where the representation principles have already been presented, this chapter reminds first some basic definitions.

8.2 DATABASES AND REAL-WORLD

Grossly speaking, a database is a representation (i.e. a model) of a part of the real world, the so-called real-world or application domain (see chapter 2). In most general cases, several databases exist simultaneously, each of them modelizes a specific part of the real-world.

The relationships between the databases and the parts of the real world they describe can be sketched according to the following figure.

Let RW be the real-world in concern, RW-A the part of this real-world described by database DB-A, and RW-B the part described by database DB-B. The following subareas can be put forward :

- some part of the real world is not covered by any database : RW-0;
- some part of the real world is covered by several databases : RW-AB;
- some parts of the real world are covered by one database only.

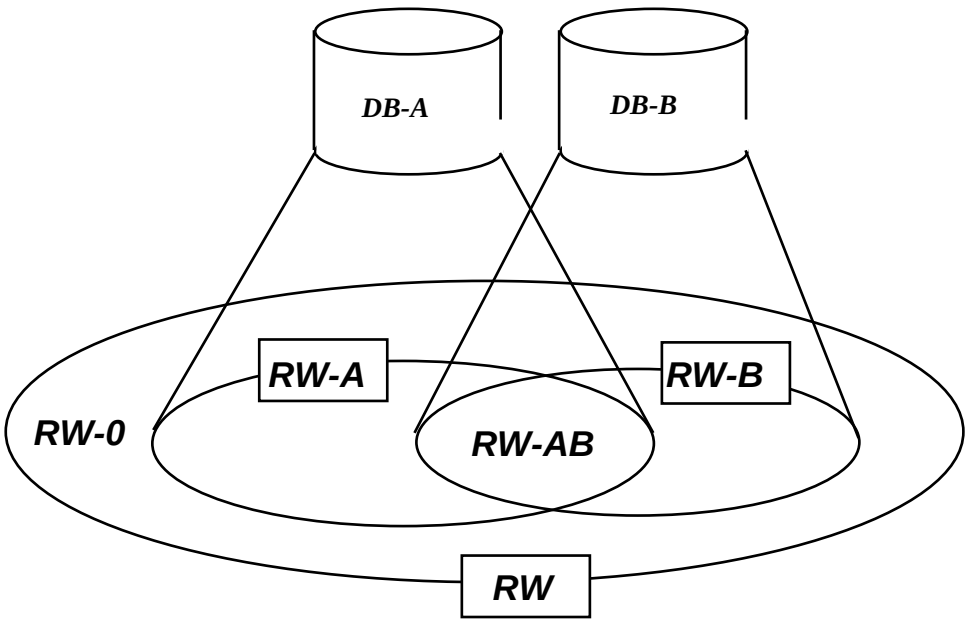


Figure 8.1 - *Each database describes a subset of the real world*

The situations that can be observed in actual applications are sometimes simpler :

- the parts covered by the databases do not overlap (i.e. RW-AB is empty);
- the part covered by one database is entirely included in the part covered by another database (e.g. RW-A includes RW-B).

In order to simplify the analysis and problem-solving approaches, we will adopt the following hypothesis.

> **There is no data sharing between databases**. In other words, even when two databases share some common descriptions of the real world, their data instances belong to one database only. Should a file be attached to more than one database, these databases will be considered as one unique higher-level database, the schema of which being the integration of the schemas of the member databases.

8.3 REAL-WORLD FACTS AND DATA

As far as the user perception is concerned, the real world is made up of facts that can be classified into fact types.

Each fact is represented in the database by some data item. A data item is an instance of a data type.

By composing these relationships, we can consider that (at least some) data types are representions of fact types.

These structures are illustrated in the following figure, that uses the same conventions as those of figure 2.4.

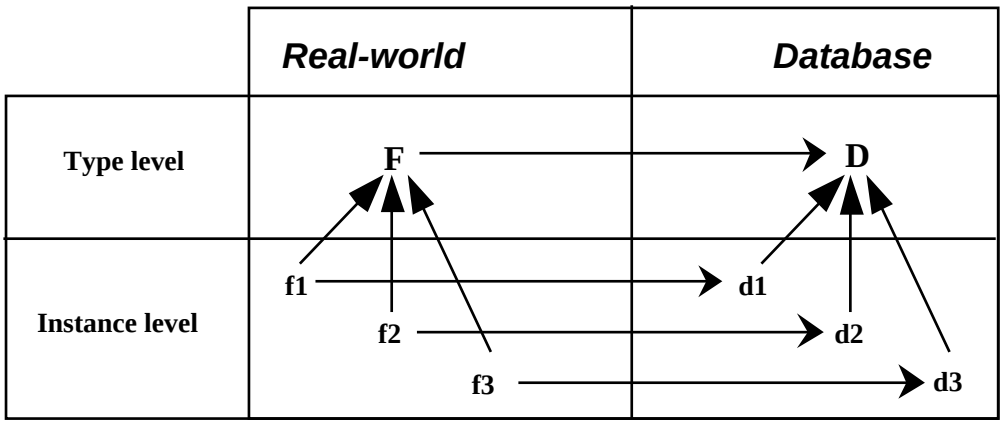


Figure 8.2 *- The relationships between types and instances, and between the real world and the database*

8.4 DATABASE ORGANIZATION

8.4.1 Database = data + schema

Any database is made up of the data part and the schema part. The schema part collects the data types and their properties. Since several schemata will be considered in the following sections, we shall call this schema the global schema of the database.

The global schema is not always materialized. In low-level data management systems (DMS) for instance (such as standard file manager) the definition of this schema is not to be found in the files themselves, but is rather spread in the user programs.

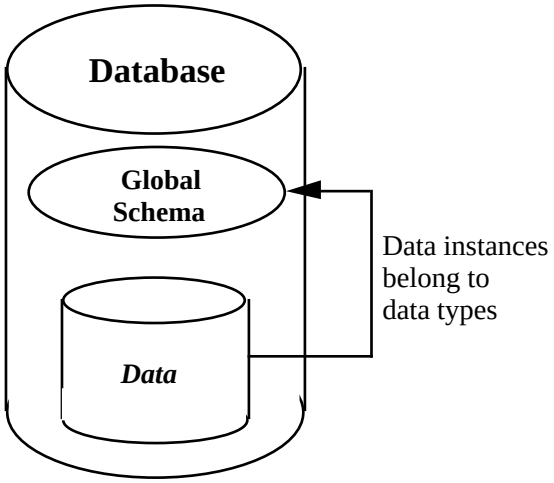


Figure 8.3 *- The parts of a database : the schema and the data set*

8.4.2 Database, views and users

Users of a database and, among them, application programs, generally have only a partial perception of the data. This perception is called a view. A view derives from the global schema, of which it is a subset, possibly restructured. A view can be shared by several programs. Basically, a view is a special kind of schema.

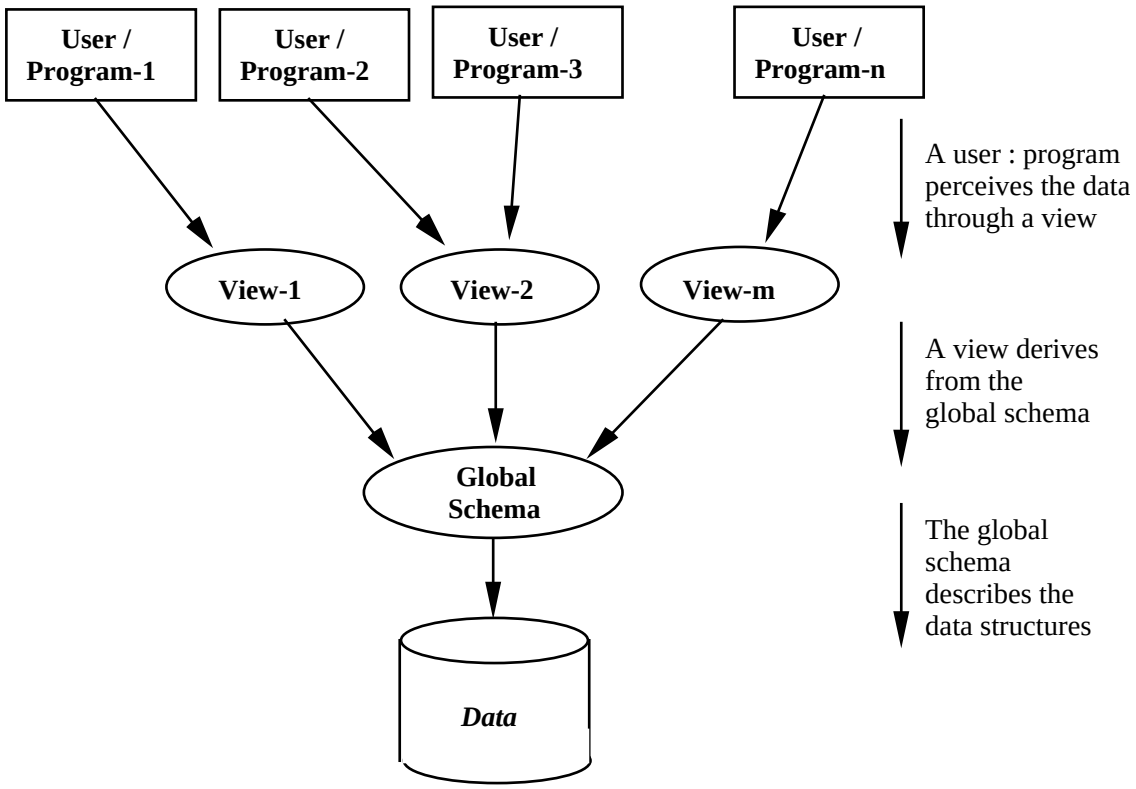


Figure 8.4 *- The relationships between the programs, the views, the schema and the data.*

A view is the complete description of the data structures of a database that are known to the user or the program. Hence, each program has one view only for each database.

In several DMS (generally DBMS), some support exists for the concept of view. However, we shall see that this support is not always sufficient in actual situations.

In some environments, programs are allowed to operate on several databases. In the framework defined here above, there will be one view per database. For reverse engineering these databases, we suggest to process each view independently. Indeed, the primary focus is on processing each database through its different schemas. Integrating these databases will be conducted in a next step.

8.4.3 Views in actual applications

The notion of view developed in the previous section is mainly conceptual. The way it is practically implemented in application programs is highly dependent on the DMS, on the host language and, above all, on design decision and programming practices (i.e. human factors).

In most general situations, the view is made up of two parts :

the explicit part includes the data structures known by the DMS, and specified in specific program sections dedicated to file and database structures declaration (DDL text, file definition statements, etc)

the implicit part includes mainly data structures of the host program in which data extracted from (or data to be stored into) the database are stored. This implicit part of the view is often very important when compared with the explicit part available. The main problems of making this part explicit can be classified as follows :

- there is no easy way to find out these data structures, except by following the control flows of the program;
- these data structures are plain variables of the host program; therefore, they can be used freely, without any concern as to their intended meaning : the same variable can be used to stored record of different types, or to store non-database values;
- each host program data structure can give partial information only on the view; using hidden or masked field is a common practice;
- data structures that seem to describe the same objects appear to be contradictory;
- in some situations, important structures can be ignored by host program data structures, but can be elicitated at the user level only; for instance, the fact that ADDRESS is a compound value can be ignored by the DMS and by the application programs; only users are aware of this substructure.

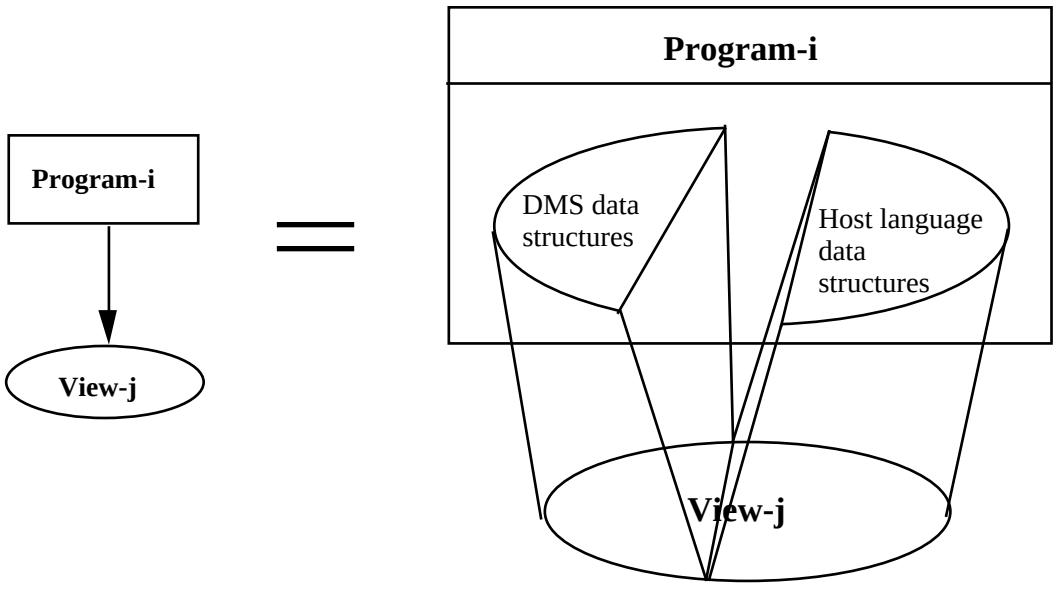


Figure 8.5 *- Zooming on the relationship between a program and a view. The expression of the latter is distributed among DMS specifications and host program specifications*

Some examples :

- an SQL column represents a compound field by storing the concatenation of the component values; the target PL/1 record includes a decomposition structure for these column values;
- a COBOL record type is declared as containing a FILLER field only; this field is given its true structure in the LINKAGE SECTION of a sub-program working on record read from the file in the calling program;
- an IMS segment (or CODASYL) record type includes a simple field that is intended to store a list of PHONE-NUMBER values. Segments/records read from the database are stored in COBOL variables that redefine this field through OCCURS specification to make the multivalue field explicit.

8.5 FACT TYPES REPRESENTATION IN MULTIPLE DATABASES

This section analyses the ways in which fact types are represented in one or several databases.

8.5.1 General case

The main situations that can be observed are illustrated in the following figure.

DB-A and DB-B represents two databases while RW represents the real world in concern. F1 to F4 are fact types. A11, A12, A2 and A5 are data types from the global schema of DB-A. B3, B21 and B22 are data types from the global schema of DB-B.

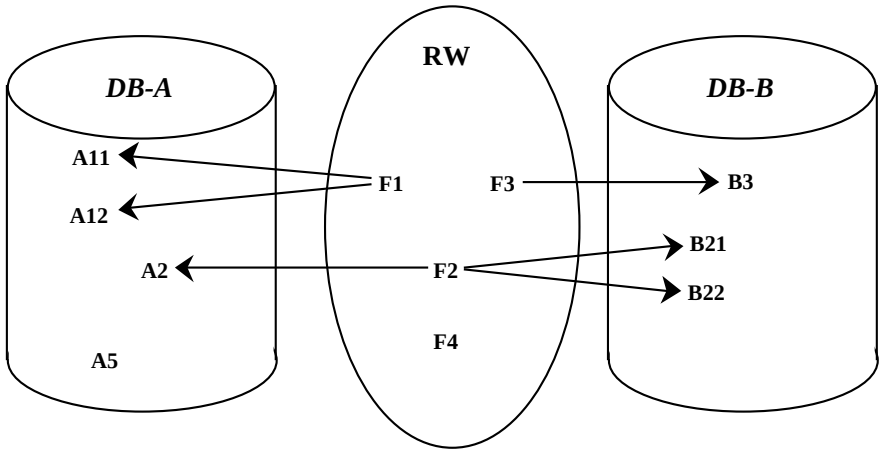


Figure 8.6 *- Real world fact types described by two databases*

According to this illustration, we can put forward the typical situations as follows :

- some data types represents fact types, while others (A5) does not represent any fact types. Such data types correspond to technical data such as program indicators, state indicators, recovery data, etc.
- some fact types are represented, while others are outside the scope of the databases (F4). If these fact types are in the application domain to be modeled, their representation should, sooner or later, be included in the final conceptual schema through semantic enrichment.
- some fact types are represented in one database only (F1,F3) while others are represented in several databases (F2).
- in a given database, some fact types are represented by one data type only (F2 in DB-A, F3 in DB-B) while some others are represented more than once (F1 in DB-A, F2 in DB-B).

In the following of this chapter, we shall ignore fact types without representation (F4) and data types that represent no fact types (A5).

8.5.2 Special cases

This section will enumerate and describe some typical situations derived from the general framework proposed hereabove. Note that, until now, we are not concerned with data distribution yet. In particular, the problem of data redundancy is left aside and will be analysed later.

8.5.2.1 Each fact type is represented only once in each database

In each database, there is no structural redundancies. However, several databases may describe the same fact types. This situation makes the reverse engineering of each database easier. However, integrating the databases must solve multiview problems.

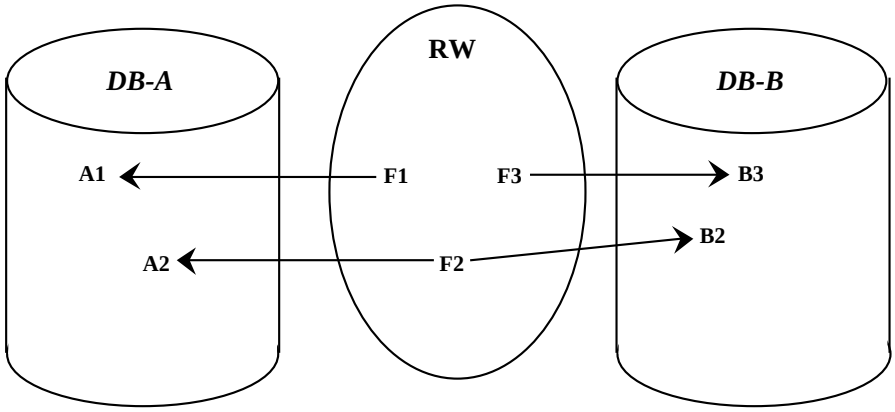


Figure 8.7 *- No fact type is represented more than once in each database*

8.5.2.2 Each fact type is represented in one database only

The set of the databases defines a partition in the fact types. This situation is ideal as far as database integration is concerned : the integrated database is the mere juxtaposition of the source databases. However, there can exist some structural redundancy problems to be solved when reverse engineering each database.

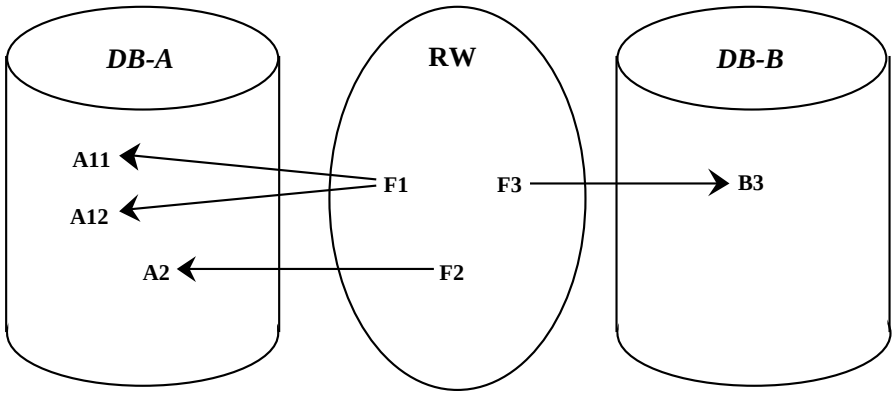


Figure 8.8 *- Some fact types are represented more than once in some databases*

8.5.2.3 Each fact type is represented only once

No database overlapping and no structural redundancies lead to the most simple situation in reverse engineering. Note that some important problems still remain to be solved : denormalization data redundancies and finding the global schema of each database.

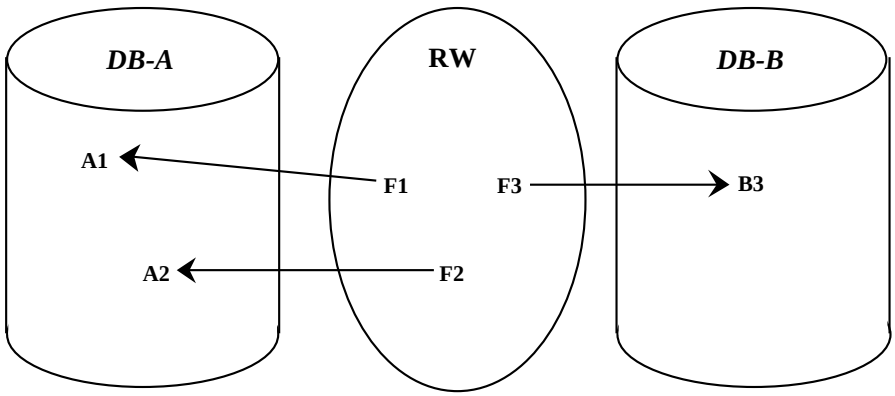


Figure 8.9 *- A fact type is represented only once*

8.5.2.4 Each fact type represented in DB-B is represented in DB-A

As far as database integration is concerned, the included database DB-B can be dropped. However, this conclusion is not true as far as data are concerned. Indeed, some facts can be represented in the included database DB-B and not in the including one DB-A. More on this topic in the next sections.

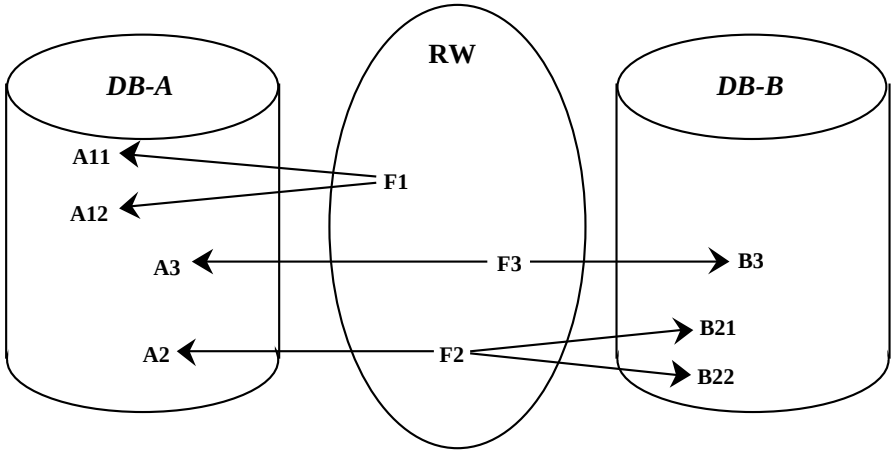


Figure 8.10 *- A fact type can be represented several times*

8.6 DATA REDUNDANCY IN SINGLE DATABASES

This section is dedicated to the analysis of the various cases of data redundancy that may hold in a database.

Let's recall that data redundancy is a situation in which a real world fact is represented more than once in a database or in a collection of databases

The problem of data redundancy accross multiple databases is analysed in the next section (5.7). We shall classify the situations according to whether data type redundancies hold or not.

8.6.1 First case : No data type redundancy

In the first family of situations, every fact type in the real-world is represented by one data type only.

8.6.1.1 With no data redundancy

This corresponds to the ideal situation in which every fact and fact type is represented only once.

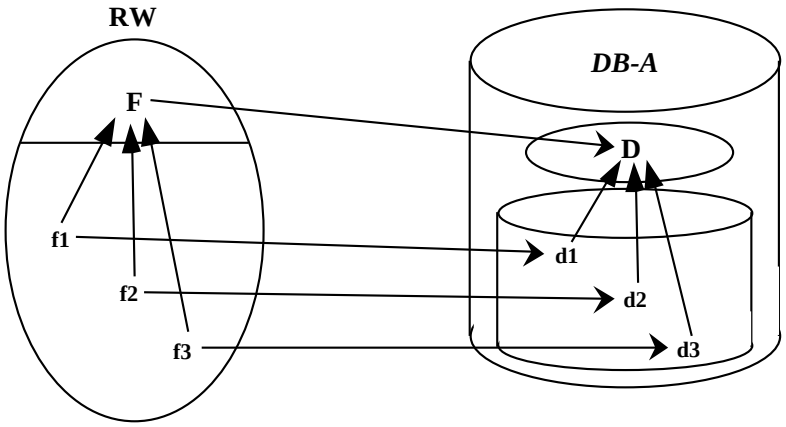


Figure 8.11 *- Each fact type is represented by one data type; each fact is represented by one data item.*

8.6.1.2 With data redundancy

Some facts, such as f2 for instance, are given more than one representation, i.e., are represented by several data instances (d21 and d22) of type D. This pattern is typical of denormalized structures (see chapter 9)

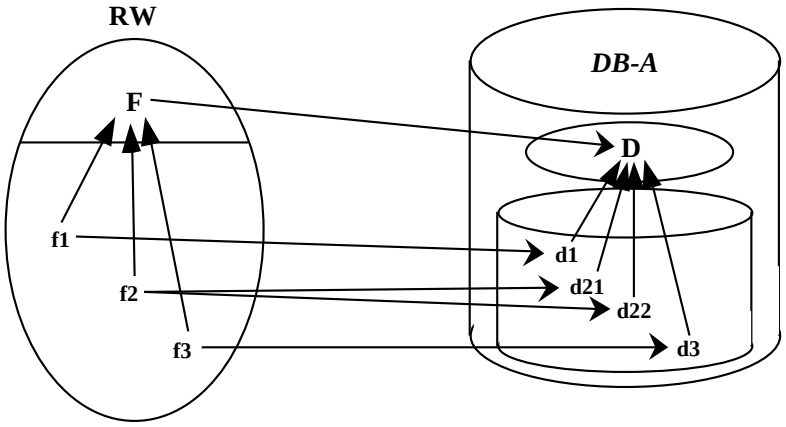


Figure 8.12 *- Each fact type is represented by one data type; some facts can be represented by more than one data item.*

8.6.2 Second case : Data type redundancy

In the second family of situations, some fact types in the real-world can be represented by several data types. We shall observe this pattern in structural redundancies (see chapter 9).

8.6.2.1 With no data redundancies

Every fact of type F is represented by one data instance belonging to one of the data type that represents F. In this situation, the set of the data types (D1 and D2) defines a partition in the facts of type F.

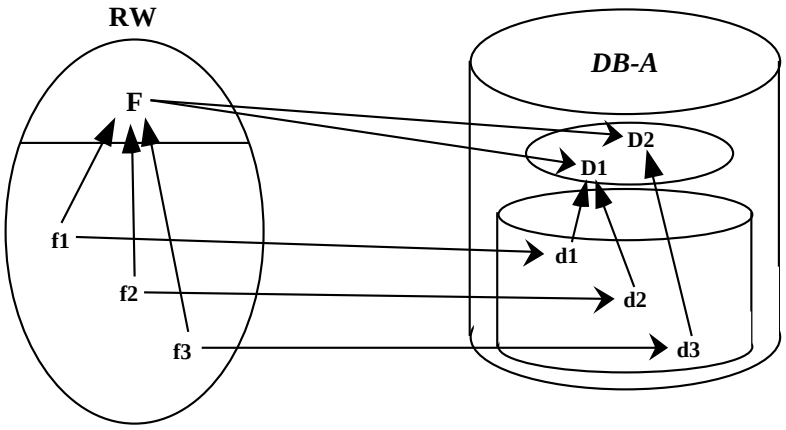


Figure 8.13 *- Some fact types are represented by several data types; each fact is represented by one data item.*

8.6.2.2 With data redundancies

There are two kinds of possible data redundancies. According to the first one, fact f of type F is represented by several data instances of the same data type, as presented in section 5.6.1.2 hereabove. The second kind is illustrated in the following figure. Fact f2 of type F is represented by data instances of several types, i.e. D1 and D2.

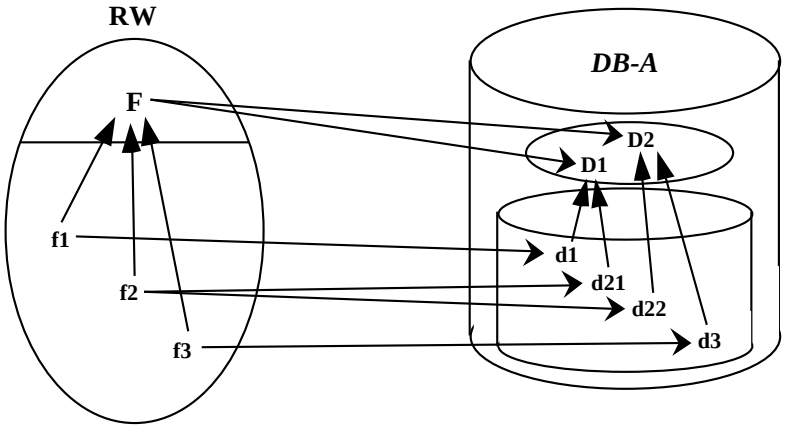


Figure 8.14 *- Some fact types are represented by several data types; some facts can be represented by several data items of different types.*

The following figure illustrates a situation in which both kinds of redundancies hold :

- fact f2 is represented by data items d21 and d22 of different types,
- fact f3 is represented by items d31 and d32 of the same type.

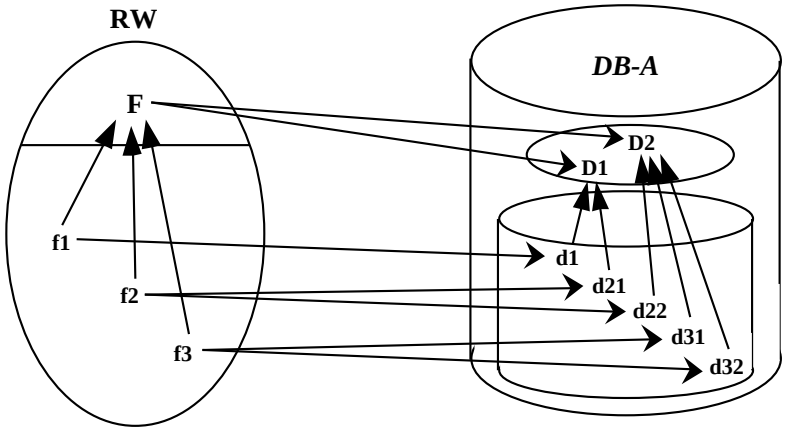


Figure 8.15 *- Some fact types are represented by several data types; some facts can be represented by several data items, possibly of different types.*

This situation is the most complex to deal with, both at the schema level and at the instance level.

8.7 DATA REDUNDANCY IN MULTIPLE DATABASES

8.7.1 Introduction

As analysed in section 5.5.1, some fact types are represented in one database only (F1 in DB-A and F3 in DB-B), while the others (F2) are represented in more than one database. The former situation relates to data redundancy in single databases, and has been analysed in section 5.6. Therefore we shall concentrate on the latter case only, where fact types are represented by data types in several databases.

In this context, two situations can be found.

- In the first one, each fact f of F is represented in one database only.
- In the second situation, a fact f of F can be represented in several database.

8.7.2 Each fact is represented in one database only

In this situation, there is no data redundancies accross the databases (but some redundancies may hold in each database). In other words, the facts of the concerned fact type are partitioned according to the databases.

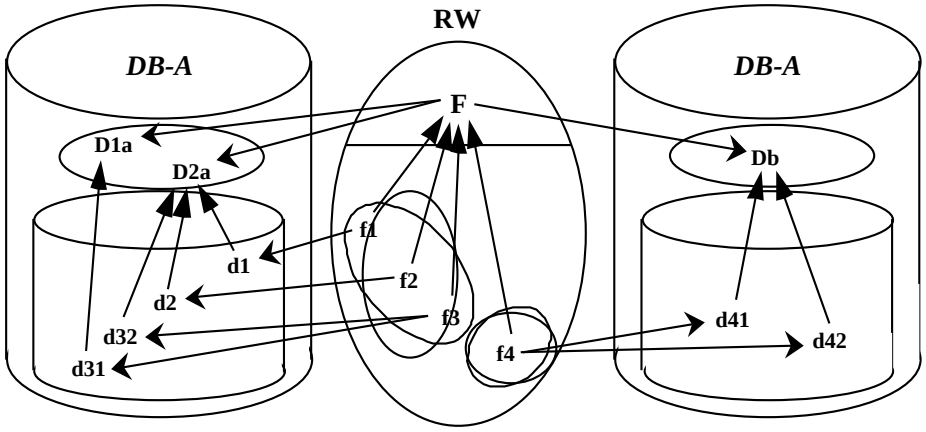


Figure 8.16 *- The real world is represented by two databases; each fact is represented in one database only (possibly several times).*

In each database DB-X, the situation can be analysed according to the criteria proposed in 5.6.

1. *There are no data type redundancies in DB-X : F is represented by data type D only.*
   a) There are no data redundancies in DB-X : each F fact is represented by one data instance d.
   b) There are data redundancies in DB-X : some F facts are represented by several data instances (this situation is illustrated by the figure above in database DB-B).

2. *There are data type redundancies in DB-X : F is represented by data types D1 and D2.*
   a) There are no data redundancies in DB-X : each F fact is represented by a data instance of D1 or D2.
   b) There are data redundancies in DB-X :
      i some facts f are represented by one data instance of D1 and one data instance of D2 (this situation is illustrated by the figure above in database DB-A),
      ii some facts f are represented by several data instances of D1 and/or data instance(s) of D2.

8.7.3 A fact may be represented in several database

This situation corresponds to redundancies accross databases. Some facts can have representations in more than one database. Note that data redundancies accross databases imply data type redundancies as well.

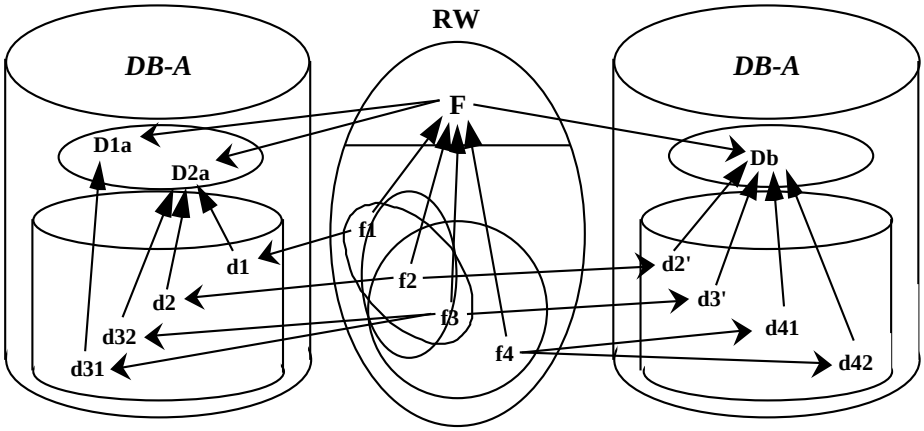


Figure 8.16 *- The real world is represented by two databases; some facts may be represented into both databases.*

In each database DB-X, the situation can be analysed according to the criteria proposed in 5.6. The different cases that can be encountered are the same as in section 5.7.3. They are recalled hereafter.

1. *There are no data type redundancies in DB-X : F is represented by data type D only.*
   a) There are no data redundancies in DB-X : each F fact is represented by one data instance d.
   b) There are data redundancies in DB-X : some F facts are represented by several data instances (this situation is illustrated by the figure above in database DB-B).

2. *There are data type redundancies in DB-X : F is represented by data types D1 and D2.*
   a) There are no data redundancies in DB-X : each F fact is represented by a data instance of D1 or D2.
   b) There are data redundancies in DB-X :
      i some facts f are represented by one data instance of D1 and one data instance of D2 (this situation is illustrated by the figure above in database DB-A),
      ii some facts f are represented by several data instances of D1 and/or data instance(s) of D2.

8.8 THE GLOBAL SCHEMA/VIEW RELATIONSHIP REVISITED

The previous sections concentrated mainly on the problem of data and data types distribution in single and multiple databases. They left aside the specific problems that concern the relationships between the global schema and the views. However, there are at the very core of the reverse engineering processes.

The problems can be synthetized as follows :

- when an explicit description of the global schema is available, it may lack some important details that can only be found in views defined on it. Let's only mention compound and multivalued fields represented as mere character strings or masked record structures.
- in many situations (with simple DMS for instance), there is no available description of the global schema. The major source of information that can lead to retrieve this schema is through the largest as possible set of views on this schema.
- as already quoted, retrieving a set of views defined on a schema is not always that trivial, specially in simple DMS that do not support them explicitly.

Comments

- The first source of problems is that some important components of a view can be translated into host program variables, and that retrieving the very structure of, say, a record type is sometimes close to a detective activity.
- The second source of problems is that each host program data structure (e.g. variable, work file, report, sort file) may bring a small part only of the structure to elicit. In some cases, it is by merging the description of dozens of host program data structure that the complete structure of a record type can be found out. In addition, some partial descriptions may be conflicting, in such a way that merging them implies conflict resolution processes.
- The third source of problems is the possibly huge number of redundant data structures descriptions that can be found. A record that is moved, directly or not, into 20 variables will be associated with at least 20 record type descriptions, possibly all identical. In addition, some data structure descriptions can be defined in several programs by including a common text in the programs which use this structure (COPY in COBOL). These descriptions bring no more information than the first one. It is necessary to manage this multiplicity.

Chapter 9

DATA REDUNDANCY

The concept of data redundancy is defined as the multiple representation of the same real-world fact. This chapter analyses in detail the concept of data redundancy, its objectives, its various forms and the ways to eliminate them when they must be discarded. Two redundancy techniques are described, namely structural redundancy and denormalization.

9.1 INTRODUCTION

The concept of data redundancy has been introduced in chaper 8, dedicated to integrity constraints. Data redundancy is an explicit violation of the database axiom : a fact is represented only once (i.e. by one piece of data only). It means that some facts may, in some circumstances, be recorded more than once in the database.

9.1.1 Objectives and drawbacks

The aim of data redundancy can be manyfold.

- The most obvious objective is to **increase the performance** of the application programs. Duplicating data in entities from where they are most often asked for is a way to decrease the number of physical accesses. Let's remind, for instance, the schema that illustrates section 2.3.11.6, and in which CUSTOMER-ADDRESS in ORDER is redundant with ADDRESS in CUSTOMER. The address of the customer of an order can be obtained without accessing the customer entity.

```
entity-type CUSTOMER(   CNUM: ..,
                        ADDRESS: ..)

entity-type ORDER   (   ONUM: ..,
                        CUSTOMER-ADDRESS: ..)

rel-type PASSES(   [0-N]: CUSTOMER,
                   [1-1]: ORDER)
ORDER.CUSTOMER-ADDRESS copy-of ORDER.PASSES.CUSTOMER.ADDRESS
```

- Duplicating data where they are the most needed **increases their availability**. This duplication will occur accross files or accross computer sites for example.
- Maintaining several versions of the same data **increases the security** of the data base. Indeed, lost data can be recovered through their copies.
- Another reason for introducing redundancies is the current **limits of the relational views**. Many relational DBMS offer the concept of view. A view is an external schema through which a user can manipulate the contents of the database. This schema gives him/her a transformed view of the actual data and is defined as the virtual result of a query. In most cases, there is no limit to the definition of views. However, views that are defined on more than one table cannot be updated (i.e. one cannot issue update queries through such views). This limitation sometimes forces the developer to propose two external schemas of the data, a schema for consultation (with multitable views), and a schema for update (with one-table views only). One way to avoid this problem is to implement unnormalized base tables that correspond to multitable views.

Redundancies implies more or less severe drawbacks that must be evaluated against their advantages.

- The most obvious problem, but the less important one, is the **increased volume** of the data.
- The second problem is the **increased update cost**. Indeed, the change of a single fact in the real-world will induce as many data updates as there are representations of this fact in the database.
- The third problem is the most important one. It derives from the fact that most DBMS offer **no support for redundancy integrity** constraints. Therefore, there is no way to enforce these constraints with a secure and centralized procedure. Taking care for updating all the copies of a piece of data is up to the programmer. Should the latter forget one update, the database is definitely corrupted. The problem can be partly controlled through access modules (see 3.7.4) which concentrate all the updates. However, this technique applies to application programs only, and brings no help for interactive users [1].

---

[1] DBMSs that offer the definition of some kind of behavioural aspects of data (the so-called *active databases*) can solve the problem to some extent : Object-Oriented databases, daemons, triggers, etc.

- The schema can be **much less readable** than its cleaned version. This makes it an increased **source of error** for low-skilled programmers, specially in maintenance tasks.

There are two main techniques to define redundancies in a schema : **structural redundancy** and **denormalization**.

9.1.2 Structural data redundancy

The schema shows explicitly constructs (data types) A and B such that the value of B can always be derived from the value of A. In other words, there exists a derivation rule (copy, formula, procedure) that gives the value of B knowing the value of A. In the example recalled hereabove, CUSTOMER-ADDRESS in ORDER (data type B) is redundant with ADDRESS in CUSTOMER (data type A). In the following example (see also 2.3.11.6) TOTAL-AMOUNT in ORDER is redundant since its values can be computed from other data values from the database.

```
entity-type PRODUCT(   PNUM: ..,
                       PRICE: ..)

entity-type ORDER  (   ONUM: ..,
                       TOTAL-AMOUNT: ..)

entity-type LINE   (   QTY : ...)

rel-type HAS-LINE (    [1-N]: ORDER,
                       [1-1]: LINE)

rel-type REF-PRO (     [0-N]: PRODUCT,
                       [1-1]: LINE)
    ORDER.TOTAL-AMOUNT copy-of  sum(for L in ORDER.HAS-LINE.LINE) of
                                L.QTY * L.REF-PRO.PRODUCT.PRICE
```

As already stated in section 2.3.11.6, a redundancy is not (necessarily) symmetrical. The redundant construct is the one which can be discarded without reducing the semantic expression of the schema.

The rule is simple :

> *construct B is redundant with construct A iff in each valid database state, any instance of B can be computed from instances of A.*

When the redundancy is symmetrical, i.e. when each construct is redundant with the other one, the construct to eliminate can be chosen arbitrarily. However, it is a good practice to keep the most general construct. For instance, choose the entity type against the attribute or the relationship type and choose the relationship type against the attribute.

The generality hierarchy can be summarized as follows (A < B means that A is less general than B) :

attribute < *relationship type* < *entity type*

9.1.3 Denormalization

In the theory of relational databases [DATE,86], normalized forms of relational schemas are presented as the best ways to represent real-world facts. The highest the normalized form the better the representation. In an unnormalized relation (or table), some real-world facts may be represented more than once, leading to data redundancy. Therefore, the relational theory proposes to decompose the unnormalized relation into two or more smaller relations being in a higher normalization form. The redundancy problem tends to disappear, but the number of resulting relations increases accordingly. In a fully normalized relation, each real-world fact is represented only once.

As opposed with the structural redundancy, a schema in which unnormalization redundancy holds doesn't show two distinct redundant constructs. Instead, the problem is only visible at the data instance level : a real-world fact type is represented by one data type, but a fact of this type can be represented by more than one data type instance.

The relational theory proposes a simple criterion for normalized relations[2] :

a first-normal-form relation is normalized iff all attributes depends on the keys only

This definition is valid for flat relations, i.e. such that each attribute value is an atomic value. However, it has been extended to non-flat structures in which compound and multivalued attributes are allowed. The dependencies implied in the definition can be functional dependencies (2.3.11.4), multivalued dependencies and join-dependencies [DATE,86]. Dependencies that doesn't satisfy the key criterion are considered as bad dependencies, since they induce redundancies. Section 2.3.11.4 proposes an example of unormalized structure. This example is rewritten herebelow as a relational table definition expressed in an intuitive pseudo-code :

```
table EMPLOYEE(    ENUM: .. key,
                   ENAME: ..,
                   DEPART-NAME: ..,
                   LOCATION: ..)

DEPART-NAME --> LOCATION
```

The relational theory explains that the functional dependency is not (only) on the key, and therefore induces redundancies as analysed in 2.3.11.4. The proposed solution is to

---

2 in relational databases, the term *key* stands for *identifier*

decompose the EMPLOYEE relation into the two fragments EMPLOYEE' and DEPARTMENT as follows :

```
table EMPLOYEE'(   ENUM: .. key,
                   ENAME: ..,
                   DEPART-NAME: ..)

table DEPARTMENT(  DEPART-NAME: .. key,
                   LOCATION: ..)

EMPLOYEE'.DEPART-NAME is-in DEPARTMENT.DEPART-NAME
```

In the latter schema, all dependencies are on the keys only, and induce no redundancies. The schema is said to be *normalized*, and the decomposition process is called *normalization.*

Denormalizing a schema is the converse process : starting with a normalized schema, replace sets of relations by joining them on common attributes. In the situation described above, it means implementing and using the EMPLOYEE relation instead of the couple of relations EMPLOYEE' and DEPARTMENT.

Starting with this relational theory, it is easy to extend the concepts to higher-level models, such as Entity-Relationship models. We will replace the notion of unnormalized relation with unnormalized entity type, attribute and relationship type.

9.1.4 Mixed redundancies

To make things a bit more complex, it is quite possible to use both redundancy techniques in the same schema construct. The following example, deriving from the CATEGORY/ PRODUCT schema of section 3.4.3, includes both structural and denormalization redundancies.

Initial schema :

```
entity-type CATEGORY (  CODE: ..,
                        VAT-RATE: ..)

entity-type PRODUCT  (  PNUM: ..,
                        NAME: ..)

rel-type BELONGS-TO  (  [0-N]: CATEGORY,
                        [1-1]: PRODUCT)
```

Unnormalized schema :

```
entity-type CATEGORY (  CODE: ..,
                        VAT-RATE: ..)

entity-type PRODUCT (   PNUM: ..,
```

```
                          NAME: ..,
                          CODE: ..,
                          VAT-RATE: ..)

rel-type BELONGS-TO (     [0-N]: CATEGORY,
                          [1-1]: PRODUCT)

PRODUCT.(CODE,VAT-RATE) copy-of
                          PRODUCT.BELONGS-TO.CATEGORY.(CODE,VAT-RATE)
```

9.1.5 Schematic representation of the redundancy structures

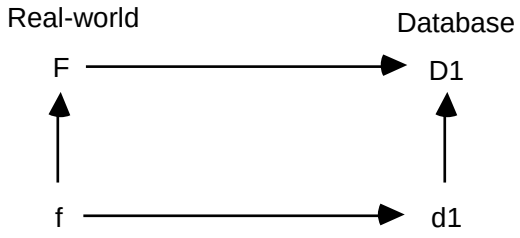
The graphical representation of the links between the real-world and database objects may help clarify the various kinds of redundancies.

We shall use the same conventions as in chapters 2 and 8. In the following schemas, let :

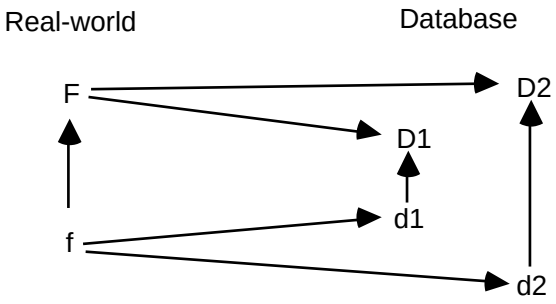
- **F** stand for a real-world fact type (such as *each department has a name*);
- **f** stand for a fact of type **F** (such as *department X has name Y*);
- **D1**, **D2**, etc stand for data types (such as *record type EMPLOYEE* or *attribute NAME*);
- **d1**, **d1i**, etc, stand for pieces of data of type **D1** (such as value *'Marketing'* of attribute *NAME*).

In addition, a vertical arc states that the bottom object is an instance of the top object (e.g. **f** is a fact of type **F**, or **d1** is a piece of data of type **D1**). A horizontal arc represents the real-world / database mapping (e.g. fact type **F** is represented by data type **D1**, or fact **f** is represented by the piece of data **d1**).

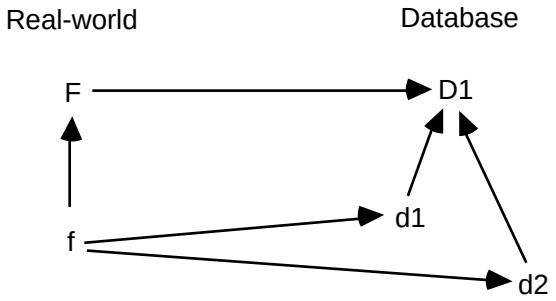
A situation **without redundancies** can be sketched as follows : the fact type **F** is represented by data type **D1**. Each fact of type **F** (e.g. **f**) is represented by one piece of data (e.g. **d1**).



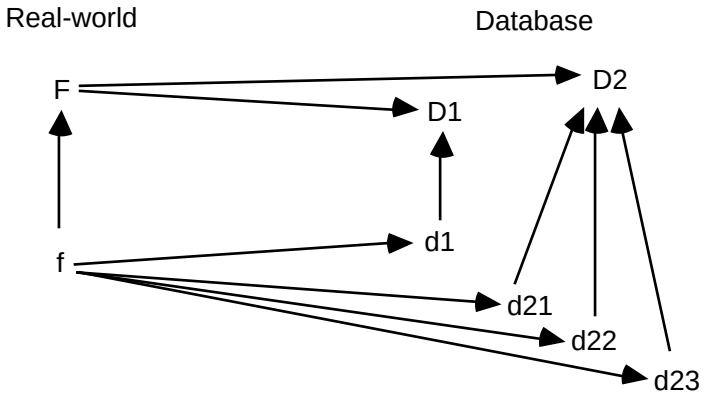
A situation **with structural redundancy** only can be represented as follows : the fact type **F** is represented by data types **D1** and **D2**. A fact of type **F** (e.g. **f**) is represented by a piece of data of type **D1** (e.g. **d1**) and by a piece of data of type **D2** (e.g. **d2**).



A situation **with denormalization redundancy** only can be represented as follows : the fact type **F** is represented by data type **D1**. A fact of type **F** (e.g. **f**) is represented by several pieces of data of type **D1** (e.g. **d1** and **d2**).



A situation **with both structural and denormalization redundancies** can be represented as follows : the fact type **F** is represented by data types **D1** and **D2**. A fact of type **F** (e.g. **f**) is represented by a piece of data of type **D1** (e.g. **d1**) and by one or several pieces of data of type **D2** (e.g. **d21**, **d22** and **d23**).



9.1.6 The data redundancy problem in Reverse Engineering

Reducing data redundancies is not a difficult problem. Structural redundancies are solved by eliminating the redundant constructs, while denormalization redundancies are solved by decomposition. These techniques will be developed in this chapter. The very problem is to detect these structures, i.e. to find out the dependencies and the derivation rules that are evidences of redundancies.

9.1.7 Data duplication

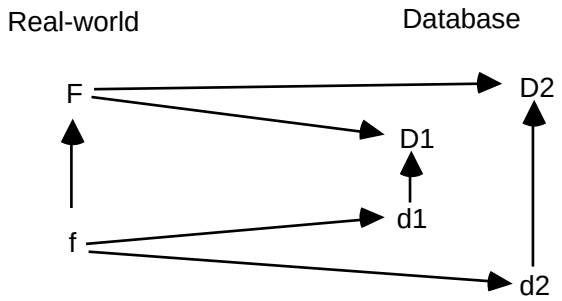
Data duplication is a special case of data redundancy that deserves specific processing in reverse engineering. For operational purposes, some parts of a database may be duplicated in the context of the same set of programs. Such is the case for archive data, print files, update files, intermediate files (e.g. sort files), input and output files (same structure but modified contents). Using such files is frequent in traditional COBOL applications, but it can be found in DBMS-based applications as well. There are basically two kinds of duplication.

1. The first case concerns **special purpose data sets** that are created by application programs either for internal use, or for communicating with other programs. They contains copies of data from the database. Their structure can be that of these data, or it can be different. *Work files*, *sort files* and *print files* falls in this category. Another important example is that of *archive data sets*.

2. The second case is that of **multi-versioned data**. In these situations, the real-world states are explicitly represented by database instances existing simultaneously. Each database state transition is carried out by producing the next database instance from the previous instance. The state changes are described in some kind of update file. This way of updating a database is typical of traditional batch data processing : *old-file + update-file --> new-file.*

   In case of large volumes of data, two versions only may be kept, namely the old state and the new state of the last state transition. It is obvious that most facts will be represented in both versions of the database.

The schematic representation is that of structural redundancy :

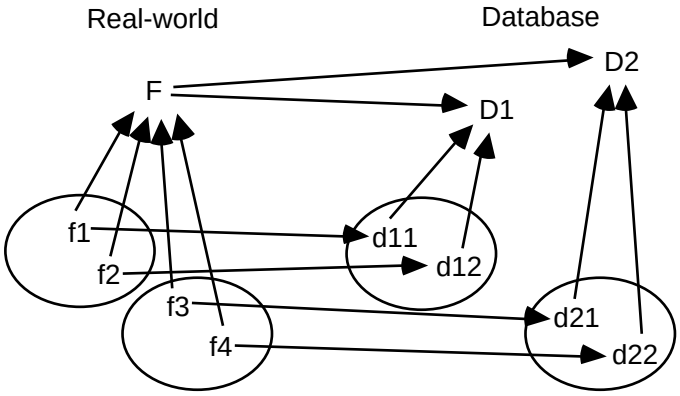


Detection of data duplication can be a non trivial task. Hints can be obtained by analysing the data flows between programs, and the data and control flows inside each program. In general, the description of duplicated data gives no new knowledge and can be discarded. This detection and elimination can be made early in the Reverse Engineering process. Let's finally observe that data duplication is (unfortunately) compatible with other kinds of redundancies.

9.1.8 Data distribution

This phenomenon is not really a data redundancy problem, but it shares some common characteristics with data duplication. It has been felt useful to analyse it in the present chapter. The data concerning a collection of similar real-world objects can be distributed into similar but distinct data types. For instance, employee data are stored into the MALE-EMPLOYEE and FEMALE-EMPLOYEE files; expenses data are distributed into the JANUARY-EXPENSES, FEBRUARY-EXPENSES, ... files. In case of pure distribution (i.e. there is no duplication), the data are **partitionned** across similar data types.

Data partitioning can be described by the following schema :



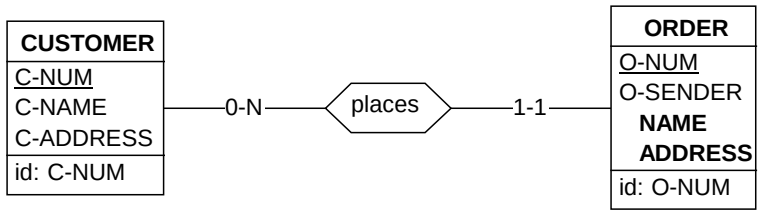
Detecting and reducing distributed data is basically a semantics-based process. Indeed, the distribution criterion is in itself a implicit semantic information (sex of employees, month of expenses). Merging these data types can be associated with the definition of a new attribute (SEX, MONTH) or of a supertype/subtypes structure.

9.1.9 Organization of the chapter

Sections 9.2 to 9.6 are dedicated to the main structural redundancy patterns, namely Attribute/Attribute, Attribute/Role, Attribute/Relationship type and Entity-type/Relationship type. Section 9.7 compares redundancy and inclusion integrity constraint, two concepts that are closely related. Section 9.8 presents an extension of the redundancy concept, the partial redundancy. Section 9.9 develops the problems of unnormalized structures, both in flat and non-flat structures. Section 9.10 evokes some other sorts of redundancies

9.2 ATTRIBUTE/ATTRIBUTE REDUNDANCY

9.2.1 Typical example




```
ORDER.O-SENDER.(NAME,ADDRESS)
                copy-of ORDER.passes.CUSTOMER.(C-NAME,C-ADDRESS)
```

Observation : NAME and ADDRESS of ORDER are duplicates of C-NAME and C-ADDRESS of CUSTOMER through *places*.

9.2.2 Description of the structure

- Attribute B is said to be redundant with attribute A through P.
- The parent PB of attribute B is an entity type, a relationship type or an attribute.
- There exists a path P from PB to A such that for each instance pb of PB a number N of values of A are connected to pb through P.
- For each instance pb of PB, the values of B are the N values of A connected to pb through P.
- A path is either a link or the composition of links.
- A link (L,R) exists between

  - an attribute and its parent (attribute, ET or RT)

  - a role and its relationship type.

  A link K has a left-side object (denoted by L(K)) and a right-side object (denoted by R(K)).

  Examples :

  if purchase is a role of RT BUYS, then (purchase,BUYS) and (BUYS,purchase) are links;

  if NAME is an attribute of ET EMPLOYEE, then (EMPLOYEE,NAME) and (NAME, EMPLOYEE) are links.

- A composition of links is made up of a sequence of links such that

  - all links are distincts,

- for any two successive links Ki and Ki+1, L(Ki+1)=R(Ki).

In this equality, an entity type and a role it can play are considered as identical objects.

Note : to simplify the notation, the path (L1,R1).(L2,R2). .. .(Ln,Rn) can be written as L1.L2. .. .Ln.Rn

In the example above,

- B = NAME of O-SENDER of ORDER,
- PB = O-SENDER of ORDER,
- A = C-NAME of CUSTOMER,
- P = O-SENDER.ORDER.passes.CUSTOMER.C-NAME

9.2.3 How to detect the problem

Heuristics :

- The redundant attributes are generally defined for efficiency purposes. Access to these attributes replaces expected accesses to other entities.

  Example : the invoicing module doesn't ask for the customer data of the order, though the latter seem necessary (according to domain knowledge).

- Ideally, they are updated together with their origin.

  Example : updating the address of a customer is followed by updates in the orders of the customer.

- This structural redundancy often induce denormalization redundancy. This means that updating an A (or B) value implies several updates of B values.

  Example : updating the address of a customer is followed by several updates in the orders of the customer.

- They are given names that evoque their origin.

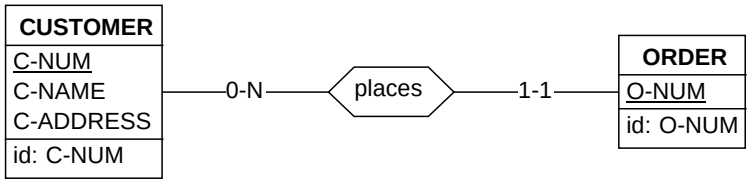
  Example : NAME and C-NAME.

- Most often, B is a single-valued attribute. Therefore, path P is many-to-one and can be found by examining many-to-one relationship types.

  Example : all the links of the path between O-SENDER and C-NAME are many-to-one.

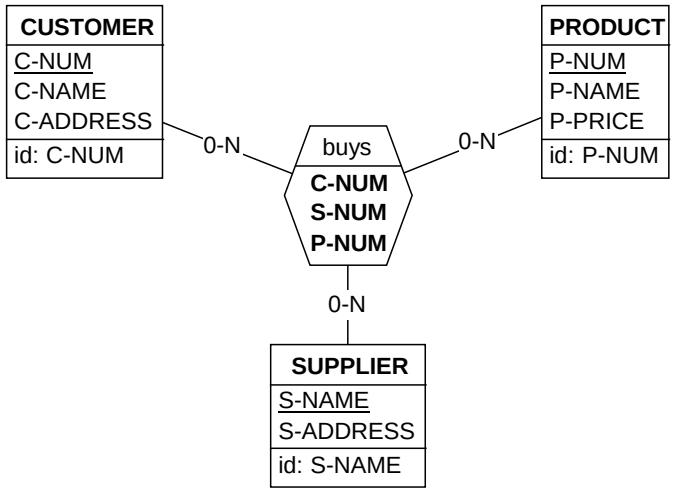
9.2.4 Redundancy elimination

The redundant attributes are discarded. If this elimination leaves a compound attribute without components, this attribute is eliminated as well. The following schema shows the result of elimination of attributes O-SENDER.(NAME,ADDRESS) of ORDER.



9.3 ATTRIBUTE / ROLE REDUNDANCY

9.3.1 Typical example



buys.c-NUM *copy-of* buys.CUSTOMER.C-NUM, etc

Observation : in any relationship buys, the C-NUM value designates the CUSTOMER entity involved in this relationship. The role of P-NUM and S-NAME is similar.

9.3.2 Description of the structure

A set of attributes of relationship type R is used to reference entities E that have a role in R.

Attribute B is said to be redundant with role S in relationship type R

- Attribute B is said to be redundant with role S in relationship type R
- B is a copy of attribute A
- PA is the entity type of A
- A is an identifier of PA.
- PA plays role S in R.

In the example above,

- R = buys
- S = CUSTOMER
- B = C-NUM of buys,
- A = C-NUM of CUSTOMER,
- PA = CUSTOMER

9.3.3 Notes

1. In the current state of the technology, this situation can only be found in conceptual schemas.
2. In some (rather older) conceptual formalisms, a relationship is defined as a t-uple of identifier values. They imply a systematic use of this kind of redundancy.
3. By applying the Relationship-type / Entity-type transformation (see 7.5.2), this redundancy is replaced with Attribute/Relationship type redundancies.

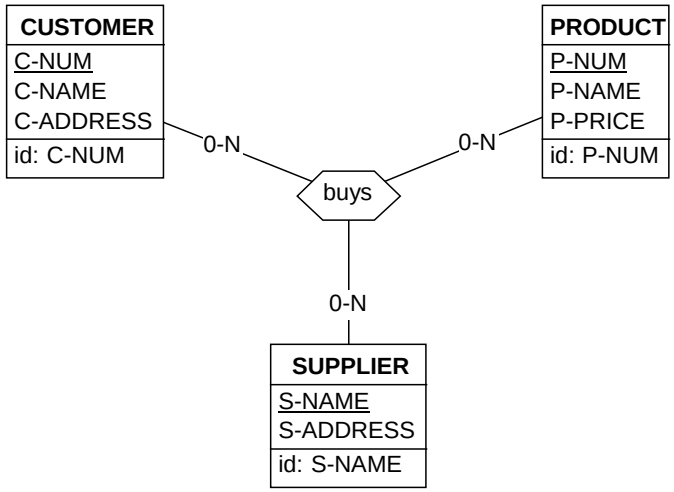
9.3.4 How to detect the problem

Attributes such as B are often given names that evokes

- either the origin entity type (e.g. REF-CUSTOMER) or the origin attribute (e.g. C-NUM),
- their reference role through prefiw or suffix such as -ref or -id (e.g. REF-CUSTOMER).

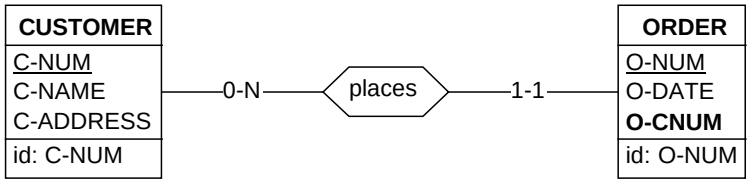
9.3.5 Redundancy elimination

The redundant attributes are discarded. If this elimination leaves a compound attribute without components, this attribute is eliminated as well. The following schema shows the result of elimination of attributes C-NUM, P-NUM,S-NAME of buys.



9.4 ATTRIBUTE / REL-TYPE REDUNDANCY

9.4.1 Typical example




```
for each o in ORDER, o.places.CUSTOMER = CUSTOMER(C-NUM = o.O-CNUM)
```

Observation : attribute O-CNUM of ORDER identifies the CUSTOMER entity linked to ORDER via *places*.

9.4.2 Description of the structure

A set of attributes of entity type E1 is used to reference entities E2 that, in addition, are linked to entities E1. The situation is similar to the attribute/attribute redundancy, except that attribute(s) A identify entity type

- Attribute B is said to be redundant with relationship type R
- PB is the entity type of B.
- B is a copy of attribute A
- PA is the entity type of A
- A is an identifier of PA.
- There exists a path P from PB to PA consisting of a relationship type or of the composition of relationship types.
- For each instance pb of PB, the values of B are the identifier values of the N PA entities connected to pb through P.

In the example above,

- B = O-CNUM of ORDER,
- PB = ORDER,
- A = C-NUM of CUSTOMER,
- PA = CUSTOMER
- P = passes

9.4.3 Notes

1. If E1 and E2 are linked through a path P made up of one relationship type only, this is a case of symmetrical redundancy[3] : either construct can be derived from the other. Theoretically, elimination of either construct can be accepted. However, in a reverse engineering context, the most explicit structure must be kept, i.e. the relationship type.
2. This situation could be compared with the attribute/attribute redundancy. The main difference is that the duplicated attribute brings no useful information but the identification of the related entity. The O-CNUM value is only a way to get the CUSTOMER entity.
3. This situation is a degenerated form of Attribute/Role redundancy.
4. This kind of redundancy is very common in network databases, such as CODASYL and TOTAL/IMAGE. In the latter, it is even mandatory : a detail variable entry data set must have both a reference attribute and an access path.
5. This kind of redundancy is also typical in some CASE tools (Silver-Run) or 4GL environments (UNIFACE).

---

[3] as demonstrated by the existence of a semantics-preserving transformation between both constructs (see Chapter 7)

9.4.4 How to detect the problem

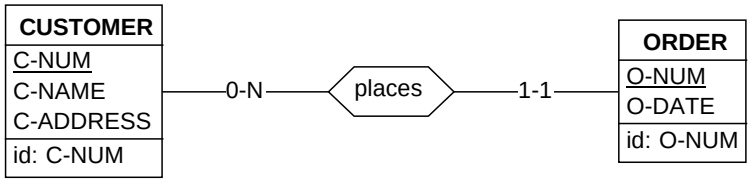
Heuristics :

- Every access path in a TOTAL/IMAGE schema induces such a redundancy.
- In CODASYL databases, the identifier of a record type cannot be made of more than one role. It is common practice to replace some roles by the identifier attributes of the owner record type. Such hybrid identifiers can be found in set type declarations : member definition with "duplicates not allowed for ..." clause.
- In some cases, attributes such as B are introduced for satisfying DMS constraints. They are therefore useless as far as access algorithms are concerned.
- Attributes can be used to bypass intermediate entity types. For instance, let's consider the following situation : an EMPLOYEE works in a SERVICE which is in turn in an ADMINISTRATION. EMPLOYEE entity type can be given attribute ADM-NAME, which designates the ADMINISTRATION of the SERVICE he works in. Here, the attribute is redundant with the composition of relationship types.
- Attributes such as B are often given names that evokes,
  1) either the origin entity type (e.g. REF-CUSTOMER) or the origin attribute (e.g. C-NUM),
  2) their reference role, through prefix or suffix ref , id, etc (e.g. REF-CUSTOMER).
- Most often, B is a single-valued attribute. Therefore, path P is many-to-one and can be found by examining many-to-one relationship types.
- Ideally, they are updated together with their origin.

  Example : changing the customer of an order is followed by updates in the O-CNUM value of the order.
- This structural redundancy often induce demormalization redundancy. This means that updating an A (or B) value implies several updates of B values. Example : updating the number of a customer is followed by several updates in the orders of the customer.

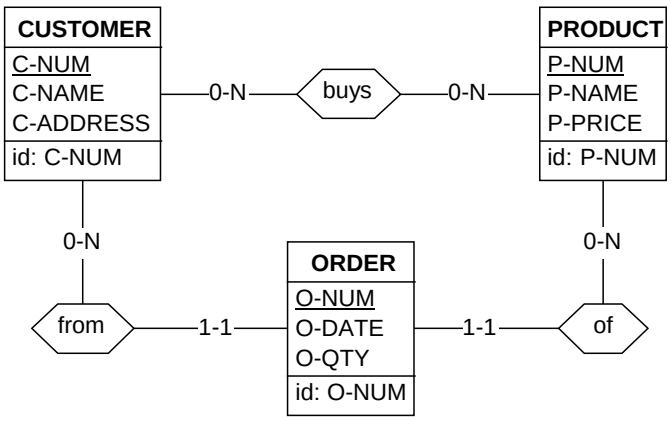
9.4.5 Redundancy elimination

The redundant attributes are discarded. If this elimination leaves a compound attribute without components, this attribute is eliminated as well. The following schema shows the result of elimination of attributes O-CNUM of ORDER.



9.5 ENTITY TYPE / REL-TYPE REDUNDANCY

9.5.1 Typical example




```
buys = from.of
```

Observation : the products a customer buys are the products of the orders of this customer. Therefore, the buys relationships can be derived from the ORDER entities. Relationship type buys is redundant with entity type ORDER.

9.5.2 Description of the structure

Two entity types E1 and E2 are connected by relationship type R. In addition, there exists entity type E, directly connected to E1 through relationship type R1 and to E2 through relationship type R2.

Composing R1 and R2 provides a virtual relationship type (or path) P that is semantically equivalent with R. In other words, there is a one-to-one mapping between the instances of R and the instances of R.

R is said to be redundant with path P.

When R1 and R2 are many-to-one, then the one-to-one mapping exists between the instances of R and the instances of E (as in the illustration above).

In this example,

- E1 = CUSTOMER
- E2 = PRODUCT

- R = buys
- E = ORDER
- R1 = from
- R2 = of
- P = CUSTOMER.from.ORDER.of.PRODUCT

9.5.3 Notes

Such a situation may occur between more than two entity types, E1, E2, .., Ei. In this case, entity type E is linked to E1, E2, .., Ei, while R has roles on E1, E2, .., Ei., or on a subset of them. The problem detection and redundancy reduction rules are adapted accordingly.

Another extension consists in considering that the path P is made up of more than two relationship types. While there is no longer a strict one-to-one mapping between E instances and R instances, the redundancy reduction rules still apply.

The case in which R is itself a path of more than one relationship type doesn't fit into this category, due to ambiguity problems that will not be studied in details, but which will be evoked in section 7 of this chapter.

9.5.4 How to detect the problem

Heuristics :

- A relationship type and an entity type have identical or similar names,
- A sequence of relationship types is very often used as an access path, therefore, a bypass may have been implemented.
- There is several ways (i.e. paths) to navigate from an entity type to another one; these paths are perhaps redundant.

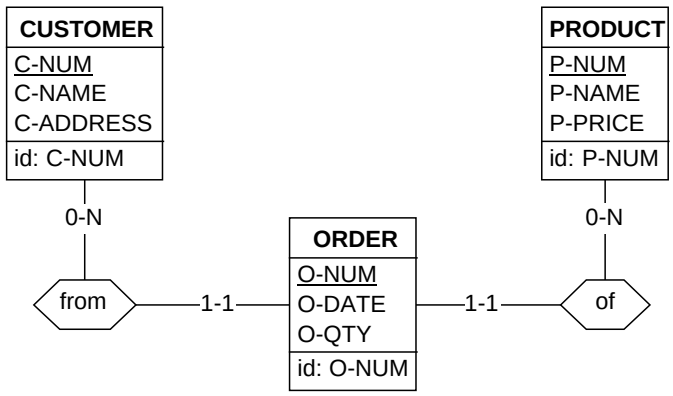
9.5.5 Redundancy elimination

The most common case concerns elimination of the relationship type. Argument : the entity type is a higher-level concept compared with the relationship type. The former will generally bear more semantics than the latter. For instance, additional attributes will be associated with the entity type instead of to the relationship type.

The elimination consists in eliminating the relationship type.

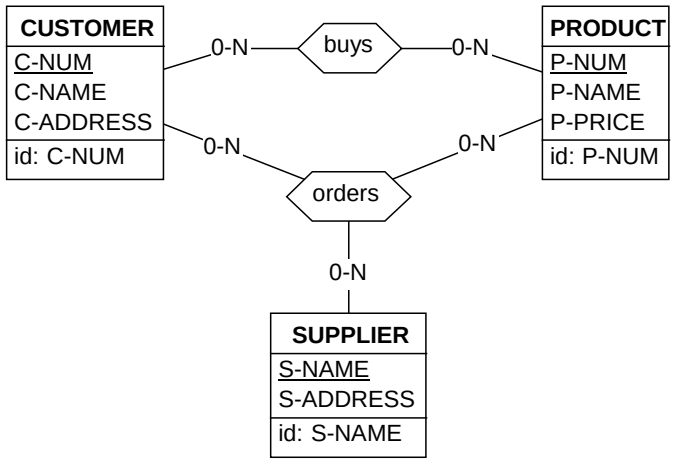
Warning : check whether this construct participates in no integrity constraint. If this is the case, refer to section 9.8.

9.5.6 Resulting schema



9.6 REL-TYPE / REL-TYPE REDUNDANCY

9.6.1 Typical example




```
buys = orders[CUSTOMER,PRODUCT]
```

Observation. The products a customer buys are the products he/she orders from suppliers. The buys relationship type can be derived from the orders relationship type.

9.6.2 Description of the structure

There exists a relationship type R between entity types E1, E2, .., Ei.

There exists a relationship type S between these entity types , or between a subset of them.

There exists a rule that allows to define the instances of S from the instances of R.

In the example above,

- E1 = CUSTOMER
- E2 = PRODUCT
- E3 = SUPPLIER
- R = orders
- S = buys
- definition rule for S = R[E1,E2]

9.6.3 Notes

In some more complex situations, R is a virtual relationship type made up of the composition of several relationship types. The rules can be easily adapted accordingly. However, detecting the actual problem can be more complicated, as discussed in section 9.7.

9.6.4 How to detect the problem

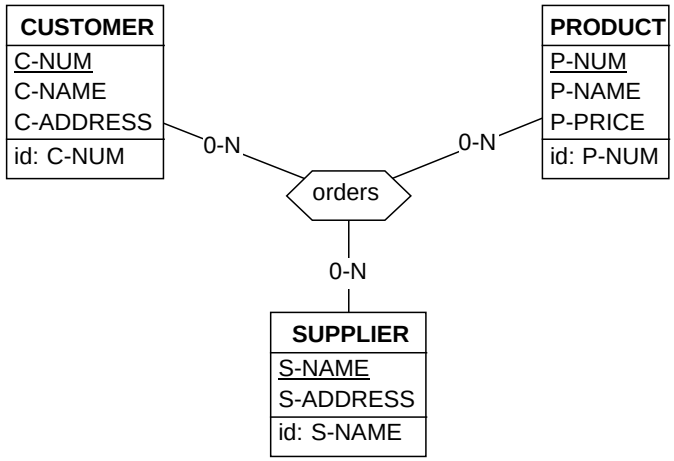
Heuristics :

- Two relationship types have similar names.
- Two relationship types have names that are semantically related (buys and orders)
- The semantics of a relationship type seems to be a subset of that of another one.
- Two relationship types are defined on a set of entity type, one of them having a lesser degree.
- In some DBMS (CODASYL for instance), a relationship type is the support of access mechanisms, such as ordering or indices, or even of integrity constraints, such as uniqueness. A way to implement different access mechanisms is to define two relationship types with the same semantics. The singular SYSTEM set is a traditional candidate for redundancy reduction in CODASYL schemas.

9.6.5 Redundancy elimination

The derived relationship type S is eliminated from the schema.

9.6.6 Final schema

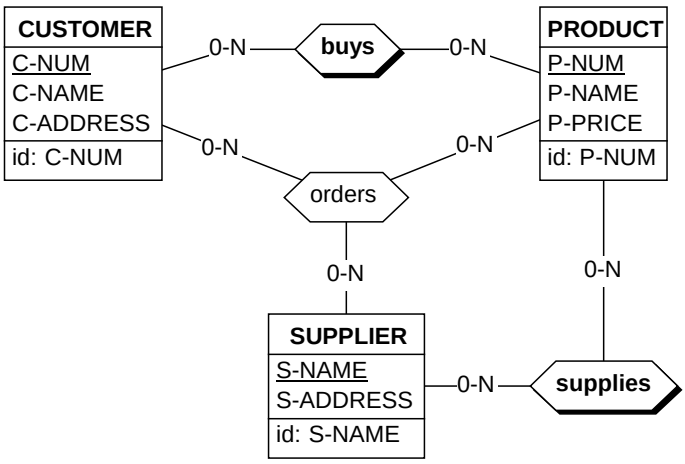


9.7 REDUNDANCYversus INTEGRITY CONSTRAINT

Similar schema patterns do not always represent similar situations. In particular, data redundancy and inclusion integrity constraints have much in common, both at the graphical representation level and at the integrity constraint level.

The following schema illustrates both concepts. The buys relationship type is a redundant construct, since each of its instances can be derived from orders instances.

However, the supplies relationship type is not redundant with orders, though orders instances share some information with supplies instances (one cannot ask a supplier for a product if this supplier doesn't supply this product). Instead, for any orders relationship, the couple [SUPPLIER,PRODUCT] must be a supplies relationship as well. This property is expressed through an inclusion integrity constraint.

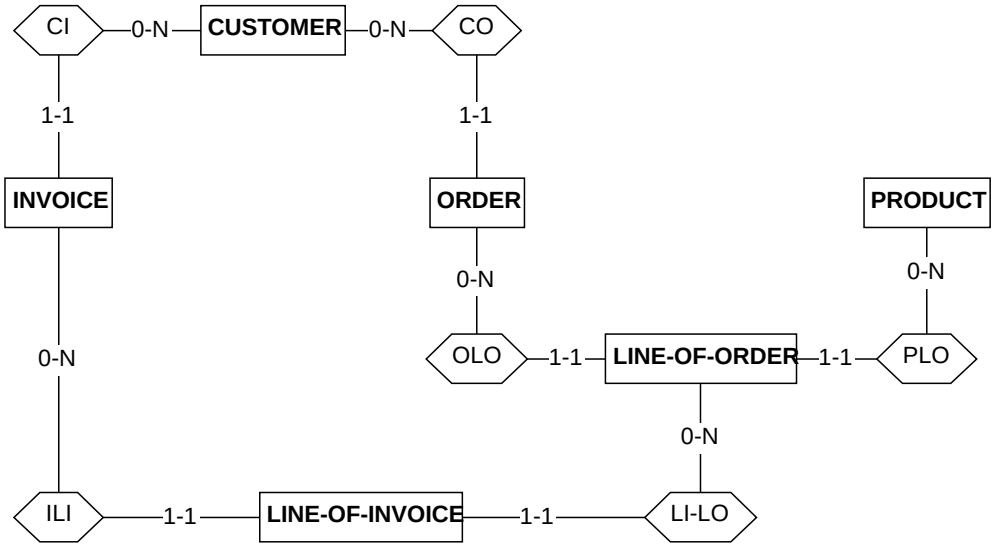



```
buys = orders[CUSTOMER,PRODUCT]
orders[SUPPLIER,PRODUCT] is-in supplies
```

Distinguishing redundancy from inclusion constraint is not always that simple.

Consider the following example that can be associated with an application domain in which,

- customers passes orders,
- orders have lines,
- each line specifies a product
- customers receive invoices
- invoices have lines,
- each line of invoice refers to a line of order.



This schema includes a circuit (`CI.ILI.LI-LO.OLO.CO`) in which we can guess that some relationship types are not quite independent of the others.

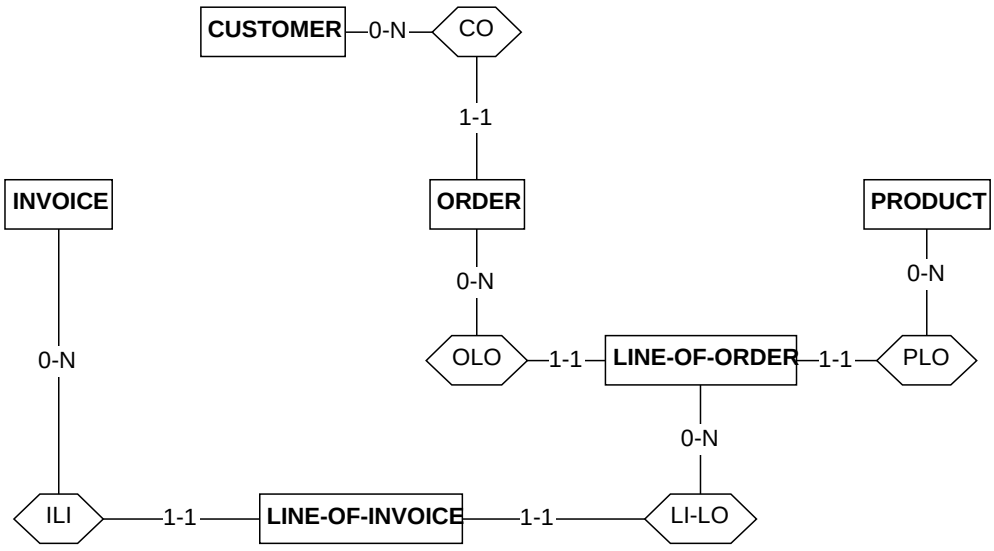
For instance, we can constrain the customer of the invoice of a line-of-invoice to be the customer of the order of the line-of-order of that line-of-invoice. This property suggests that some redundancies may hold in the schema. Finding them is not so obvious. Indeed, this property expressed hereabove can be formalized by the constraint : CI.ILI = LI-LO.OLO.CO This is clearly a redundancy expression, *but it doesn't help us in eliminating any construct of the schema*.

Let's try to explicit the relation between each relationship type and the other ones.

```
CO :      CI.ILI.LI-LO.OLO is-in CO
CI :      CI    = ILI.LI-LO.OLO.CO
ILI :ILI   is-in CI.CO.OLO.LI-LO
LI-LO :   LI-LO is-in ILI.CI.CO.OLO
OLO :OLO   is-in CO.CI.ILI.LI-LO
```

Through this analysis we can conclude that there is one redundant construct only, namely relationship type CI. All other expressions are inclusion integrity constraints.

A non-redundant version of this schema can be as follows[4].



It should then be decided whether this version is still clear enough and whether the former, redundant, schema is not to be preferred.

Indeed, a schema that doesn't make the relation between the invoices and their customers explicit would probably puzzle its readers !

---

4 As will be showed in section 9.8, this schema is still not quite equivalent to the former one

9.8 PARTIAL REDUNDANCY

The situations mentioned so far exhibited one simple redundancy pattern at a time. Moreover, when an object B was recognized redundant with object A, no subpart of B escaped this property.

However, this hypothesis is not always satisfied. More general situations can be described as follows :

1.B is redundant with A;

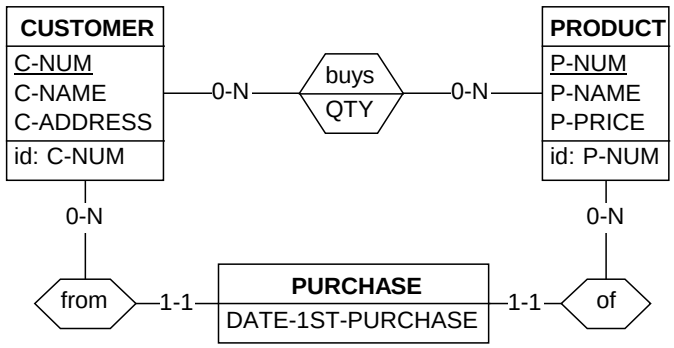
2.a subpart BB of B has no conterpart is A.

The procedures suggested hereabove cannot cope adequatly with such problems. Indeed, discarding B would lead to information loss (through BB instances).

In fact, this situation includes both redundancy and multi-view problems : some parts of instances of B are redundant with some parts of instances of A. In addition, some aspects of B are not found in A.

The following example illustrates these concepts. Relationship type buys (i.e. B) is clearly redundant with entity type PURCHASE (i.e. A). However, attribute QTY (i.e. BB) of buys has no counterpart in PURCHASE.

Therefore, we cannot simply eliminate *buys* from this schema.

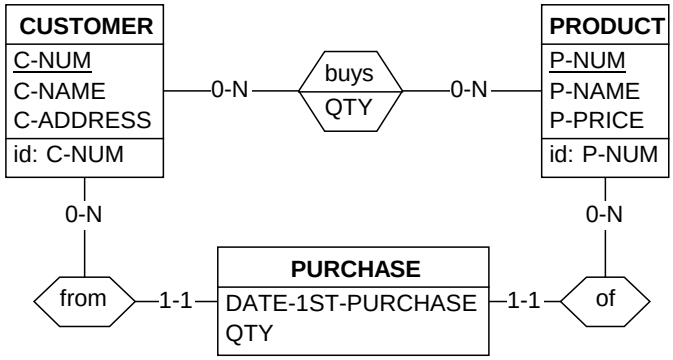


Dealing with such complex situations can be done as follows.

1.*First phase* : **view integration**

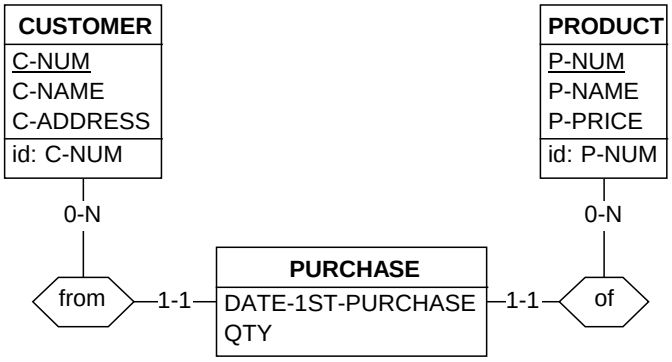
First let's consider that B and A are two views of the same real-world fact(s) and/or object(s). The fact that A and B belongs to the same schema is of no

importance here[5]. A and B are then integrated in such a way that A includes now the BB subparts of B. In this process, both A and BB can be given formats different from their original one (e.g. attribute BB can be transformed into an entity type; relationship A could be transformed into an entity type; etc). In the example given above, a new attribute QTY will be added to entity type PURCHASE. From now on, the view PURCHASE+from+of+QTY strictly includes the view buys+QTY. This integration results in the following schema :



2.*Second phase* : redundancy reduction

We are now in a standard situation where B + all its subparts are redundant with A. We are allowed to eliminate A, that is, in our example, relationship type buys with its attribute QTY. The resulting schema is as follows.



Notes

1. In some schemas, the subpart BB of B can be more abstract than merely an attribute or a role. For instance, B may be a component of an identifier or of any other integrity constraint (therefore, BB is the partnership of B in this constraint). This information must be transfered to construct A before deletion of B.

   This transfer can be simple, but it can be somewhat complex in some situations. A good way to tackle this problem is to use schema transformation (see chapter 7) in

[5] In principle, the theory of schema integration is based on a multi-view approach (see chapter 10). However, it can be extended to single view situations as well.

order to make constructs A and B more comparable : giving either A or B the same representation allows an easier transfer of characteristics between them. For instance, transforming buys into an entity type, or transforming PURCHASE into a relationship type can make the problem clearer.

2. The final schema of section 9.7 has lost an important information. Indeed, the redundancy expression

```
CI : CI = ILI.LI-LO.OLO.CO
```

states that the right-side composition has the same properties, and particularly the same cardinality constraints, as CI.

In particular,

1. every INVOICE entity is linked with at least one CUSTOMER entity;
2. an INVOICE entity cannot be linked with more than one CUSTOMER entity;
3. a CUSTOMER entity can be linked with no INVOICE entities;
4. a CUSTOMER entity can be linked with several INVOICE entities.

Properties 1, 3 and 4 can be inferred from the cardinality constraints of the components of the composition. However, **property 2 cannot**. Indeed, composing a one-to-many link (ILI) with many-to-one links (LI-LO, etc) leads to a many-to-many link (without reference to the real-world, the schema allows several CUSTOMER entities being linked to each INVOICE entity).

Therefore, **property 2 must be asserted as an additional integrity constraint**.

9.9 UNNORMALIZED STRUCTURES

9.9.1 Introduction

Unnormalization is the second major source of data redundancies. Formally, it consists in adopting a low normal form in data structures, i.e. in grouping data that are normally separate but logically connected. Section 9.1.3 of this chapter shows that the EMPLOYEE table consists of information about employees and information about department. Ideally these informations must be separate, though they are related. Restructuring unnormalized data structures in order to eliminate such awkward aggregations is called **normalization**. This process has been extensively developed in the framework of the relational theory.

Developing normalization principles for the Entity-Relationship model can best be done by translation of the results obtained in the relational theory. Therefore, this section will first recall some interesting results from this theory, then it will apply them to the E-R

structures. On the other hand, only results concerning the normalization of flat table structures (or what we shall call 1NF structures) can be explained extensively in such a manual. Since non-flat structures are very common (see COBOL record types for instance), and due to the complexity of the theory that studies them, another approach will be followed : we shall reconstruct an simplified adhoc theory from a selected collection of examples.

9.9.2 1NF normal forms in the relational theory

This section will only recall some practical principles from the relational theory. A more in-depth development of this theory will be found in the general literature on databases; see [DATE,86] or [ULLMAN,88] for example.

9.9.2.1 Relation schema in first normal form (1NF)

Relation schema R is in first normal form if its attributes are defined on simple domains, i.e. if each attribute value is a single, atomic, value. A relation schema is not in first normal form (i.e. it is in N1NF or NF2) if some attributes are multivalued and/or non atomic. A 1NF relation corresponds to a flat table representation. The standard relational model studies 1NF relations only.

9.9.2.2 Functional dependency (FD)

Let R(IJK) be a relation schema, where I, J and K are non-empty subsets of the attributes of R. Functional dependency (or FD) I-->J holds in relation R(IJK), iff in all the R tuples where value i of I appears, the same value of J appears as well, whatever i. In other words, if tuples (i1,j1,k1) and (i2,j2,k2) are in an instance of R, then i1=i2 implies that j1=j2. I is called the left-hand side (LHS) of the FD while J is called its right-hand side (RHS). We shall say that the RHS depends on the LHS.

Example :

let's consider the relation schema

```
ORDER(ONUM,DATE,CUS-NUM,CUS-NAME,CUS-ADDRESS)
```

If the values of CUS-NAME and CUS-ADDRESS depend on CUS-NUM only, we can assert the FD :

```
CUS-NUM --> CUS-NAME,CUS-ADDRESS
```

in other words, the name and the address of a customer are the same, whichever order in which this customer appears.

Some properties of FD :

- transitivity : if A --> B and B --> C then A --> C
- trivial FD : if I' is a subset of I then I --> I'
- augmentation : if A --> B, then A,C --> B
- decomposition/addition : if A --> B,C then A --> B and A --> C and conversely
- pseudo-transitivity : if A --> B and B,C --> D then A,C --> D
- the projection preserves a FD provided the LHS and a subset of the RHS are in the projection.

A FD I-->J is minimal iff there is no proper subset I' of I such that I'-->J still holds.

Let R be a relation schema, and F a set of minimal FD holding in R. A minimal cover M of F is a set of minimal FD such that

- all FD of F can be derived from M by (pseudo-) transitivity and ...,
- and no proper subsets of M enjoy that property.

Example

In the following schema,

```
ORDER(ONUM,CNUM,CADDRESS,DATE,PNUM,UPRICE,QTY)
fd1 : ONUM --> CNUM
fd2 : CNUM --> CADDRESS
fd3 : ONUM --> CADDRESS
fd4 : CNUM,DATE --> ONUM
fd5 : PNUM,ONUM --> QTY
fd6 : PNUM -- UPRICE
fd7 : CNUM,DATE,PNUM --> QTY
fd8 : ONUM --> DATE
```

the minimal cover is

```
fd1 : ONUM --> CNUM
fd2 : CNUM --> CADDRESS
fd4 : CNUM,DATE --> ONUM
fd5 : PNUM,ONUM --> QTY
fd6 : PNUM -- UPRICE
fd8 : ONUM --> DATE
```

since fd3 can be derived from fd1 and df3 by transitivity, and fd7 can be derived from fd4and fd5 by pseudo-transitivity.

9.9.2.3 Key of a 1NF relation

A key K of relation R is a subset of the attributes of R such that, for any attribute A of R, the FD K-->A holds. In other words, there cannot be more than one tuple in any instance of R with the same values for the attributes of K. Key K is minimal iff there is no proper subset K' of K such K' is still a key of K.

Example

The keys of the ORDER schema hereabove are

`(ONUM,PNUM)` and

`(CNUM,DATE,PNUM)`

9.9.2.4 1NF normal forms

The normal form is a property of a relation schema that defines the risk its instances may include redundancies (cf. 9.1.3). Several normal forms have been defined for 1NF relation schemas. They are as follows.

Second normal form (2NF)

A relation schema is in 2NF iff

- it is in 1NF
- and no non-key[6] attribute depends on a proper subset only of a key.

A 1NF schema that is not in 2NF will induce redundancies in its instances. Example of non 2NF schema :

```
ORDER-LINE(ONUM,PNUM,QTY,U-PRICE)
PNUM --> U-PRICE
```

Third normal form (3NF)

A relation schema is in 3rd normal form iff

- it is in 2nd normal form
- no non-key attribute depends on other non-key attributes.

A 1NF schema that is not in 3NF will induce redundancies in its instances. Example of non 3NF (but 2NF) schema :

---

6 Any component of a key is a key attribute. A non-key attribute doesn't appear in any key

```
ORDER(ONUM,DATE,CUS-NUM,CUS-NAME)
CUS-NUM --> CUS-NAME
```

Boyce-Codd normal form (BCNF)

A relation is in BCNF iff

- the LHS of all minimal FD are keys.

Due to its simplicity, this normal form is generally used in replacement of the other ones. It is a bit more specific than the 3NF since some 3NF relations are not in BCNF, while all BCNF are in 3NF. A 3NF schema that is not in BCNF will induce redundancies in its instances. Example of a 3NF but non BCNF schema :

```
STUDY(STUDENT,SUBJECT,TEACHER)
STUDENT,SUBJECT --> TEACHER
TEACHER --> SUBJECT
```

Other normal forms

There are other normal forms based on dependencies. The 4NF is concerned with multivalued dependencies while 5NF is based on join dependencies. They have a lesser importance in the context of reverse engineering. The reader is invited to consult standard references such as [DATE,86] or [ULLMAN,88] on this topic.

9.9.2.5 The relational decomposition theorem

The relational theory establishes that [DATE,86] :

```
R(IJK)
I-->J      ===>  R = R[IJ] * R[IK]
```

This property states that, whenever a relation schema R, the attributes of which can be partitioned into non-empty subsets I, J and K, is such that dependency I-->J holds, then it can be replaced by its projections on IJ and on IK. It establishes the condition under which a relation schema can be decomposed without loss of data. This theorem has been proposed by Casey and Delobel [CASEY,73]. A more general version, based on multivalued dependencies has been proposed by Fagin [FAGIN,77].

The following schema

```
ORDER(ONUM,DATE,CUS-NUM,CUS-NAME,CUS-ADDRESS)
CUS-NUM --> CUS-NAME,CUS-ADDRESS
```

can be replaced by

```
ORDER(ONUM,DATE,CUS-NUM)
CUSTOMER(CUS-NUM,CUS-NAME,CUS-ADDRESS)
```

Let us observe that the theorem doesn't mention any inclusion constraint between these projections. By reference with the semantics of these structures, we can add the inclusion integrity constraint :

```
ORDER[CUS-NUM] is-in CUSTOMER[CUS-NUM]
```

9.9.2.6 Relational normalization

The relational theory proposes to decompose relation schemas the normal form of which is too weak. In the framework of FD, it is adviced to try to gain BCNF, or at least 3NF.

The literature proposes various procedures for giving a relation schema R with FD set F an adequate normal form. The following is one of the most natural procedure.

1. Compute M a minimal cover of F;
2. Compute the keys of R;
3. Mark the bad FD of M, i.e. those the LHS of which is not a key;
4. Choose a marked FD of M the RHS of which is the LHS of no other FD of M;
5. Decompose R according to this FD, producing fragment R1 and R2.
6. Repeat steps 1 to 5 on the fragments until their minimal cover no longer includes bad FD.
7. If a marked FD still exists, but criterium 5 fails, then there exist some FD forming a circuit. The corresponding fragment is in 3NF, but not in BCNF. It can be kept as such or it can be decomposed further.
8. If several fragments have the same key, and if their merging (through joins) is semantically correct, merge them.

Example

Let's consider the following schema, in which the minimal cover has been computed already

```
ORDER(ONUM,CNUM,CADDRESS,DATE,PNUM,UPRICE,QTY)
fd1 : ONUM --> CNUM
fd2 : CNUM --> CADDRESS
fd3 : PNUM,ONUM --> QTY
fd4 : ONUM --> DATE
fd5 : PNUM,DATE --> UPRICE
```

the key is:

```
key : (ONUM,PNUM)
```

the bad FDs are the following : fd1, fd2, fd4, fd5

This schema will be decomposed as follows :

ORDER according to fd2 :

```
CUS(CNUM,CADDRESS)
ORD(ONUM,CNUM,DATE,PNUM,UPRICE,QTY)
```

ORD according to fd5 :

```
CUS(CNUM,CADDRESS)
PRO(PNUM,DATE,UPRICE)
OR(ONUM,CNUM,DATE,PNUM,QTY)
```

OR according to fd1, then fd4, then merging :

```
CUS(CNUM,CADDRESS)
PRO(PNUM,DATE,UPRICE)
HEADER(ONUM,CNUM,DATE)
LINE(ONUM,PNUM,QTY)
```

9.9.3 Application of the 1NF theory to E-R structures

The results recalled hereabove can be applied to E-R structures provided we can give these structures relational expressions. Such expressions are rather straigthforward in some simple, but common, situations that will be described.

9.9.3.1 Normal forms of an entity type

If entity type E has atomic and single-valued attributes only, E can be interpreted as a 1NF relation schema. The following example shows how entity type EMPLOYEE is given a relational interpretation.

```
entity-type EMPLOYEE (   ENUM : ..,
                         ENAME : ...,
                         DEPART-NAME : ...,
                         LOCATION : ...)
```

can be interpreted as relation schema

```
EMPLOYEE (ENUM,ENAME,DEPART-NAME,LOCATION)
```

This translation allows us to give E-R functional dependencies (see 2.3.11.4) a relational interpretation as well :

```
entity-type EMPLOYEE (   ENUM : ..,
                         ENAME : ...,
                         DEPART-NAME : ...,
                         LOCATION : ...)

DEPART-NAME --> LOCATION
```

can be interpreted as follows,

```
EMPLOYEE(ENUM,ENAME,DEPART-NAME,LOCATION)

DEPART-NAME --> LOCATION
```

Therefore, we can qualify the normalization state of the attributes of an entity type in the same way as we did for relation schemas : 2NF, 3NF and BCNF. The criteria for these normal forms are easily translated (replacing the term relational key by E-R identifier). For instance, the example above shows that entity type is EMPLOYEE in 2NF but not in 3NF.

Notes

1. The relational normal forms are defined in reference with the concept of key. However, the definition can still apply to entity-types that have no identifiers. For instance, the following example is still in 2NF, though the identifying attribute ENUM has been dropped :

   ```
   entity-type EMPLOYEE'(  ENAME : ...,
                           DEPART-NAME : ...,
                           LOCATION : ...)

   DEPART-NAME --> LOCATION
   ```

   It is easy to observe that this schema still leads to data redundancies. The trick is the following : two entities are always distinct, whatever their attribute values, because entities are said to be auto-identifying (i.e. the E-R model supports the object-identity property). Therefore, the relational correspondence stated above is not quite correct. A better expression would have been the following :

   ```
   EMPLOYEE (E#, ENUM, ENAME,DEPART-NAME,LOCATION)

   DEPART-NAME --> LOCATION
   ```

where E# is an internal entity identifier that ensures entity identity, independently of the attribute values.

Consequently, any entity type, even without identifiers, is represented relationally by a relation schema with at least one key :

```
EMPLOYEE' (E#,ENAME,DEPART-NAME,LOCATION)

DEPART-NAME --> LOCATION
```

2. The LHS and RHS can include not only attributes, but also related entities. This extension is similar to that of identifiers (2.3.8). It will not be developed further (see [HAINAUT,90]).

9.9.3.2 Normalization of an entity type

Unfortunately, the analogy between entity types and relation schemas does not stand as straigthforward for normalization as it does for normal form definition. Indeed, the strict application of the relational decomposition of the EMPLOYEE example would lead to the following result :

```
entity-type EMPLOYEE (
                    ENUM : ..,
                    ENAME : ...,
                    DEPART-NAME : ...)

entity-type DEPARTMENT (
                    DEPART-NAME : ..,
                    LOCATION : ...)

EMPLOYEE.DEPART-NAME is-in DEPARTMENT.DEPART-NAME
```

That's OK for relational structures, but far from natural for E-R schemas. We must therefore find another way to get rid of the normalization problem. The most intuitive solution is to make explicit the notion of DEPARTMENT as an entity type then to link it to the EMPLOYEE entity type as follows :

```
entity-type EMPLOYEE (
                    ENUM : ..,
                    ENAME : ...)

entity-type DEPARTMENT (
                    DEPART-NAME : ..,
                    LOCATION : ...)

rel-type WORKS-IN( [0-N]: DEPARTMENT, [1-1]: EMPLOYEE)
```

Note : the cardinality constraint of DEPARTMENT should be [1-N], since, according to the first schema, a DEPARTMENT entity cannot exist without being associated with an EMPLOYEE. Choosing cardinality [0-N] is a decision that gives the schema a greater generality.

A more abstract description of the normalization process is as follows :

Unnormalized entity type :

```
entity-type E (IJK)
J-->K
J is not an identifier of E
```

Normalized schema :

```
entity-type E1(I)
entity-type E2(JK)
rel-type E12([0-N]:E2,[1-1]:E1)
```

We can therefore sketch the normalization procedure as follows :

Let F be the set of FD in entity type E;

1. Compute M a minimal cover of F;
2. Compute the identifiers of E;
3. Mark the bad FD of M, i.e. those the LHS of which is not an identifier;
4. Choose a subset of the marked FD of M
   a) the RHS of which is the LHS of no other FD of M
   b) that have the same LHS; let RHS1, ..., RHSn be the RHS of these FD;
5. Define a new entity type E' with attributes in LHS, RHS1, ..., RHSn.
6. Redefine E by dropping the attributes in LHS, RHS1, ..., RHSn.
7. Define a new relationship type R between E and E' as follows :
   ```
   rel-type R([0-N]:E',[1-1]:E)
   ```

A complete axiomatization of this procedure can be found in [HAINAUT,91a]. The proposed normalization can be carried out by using the schema transformation techniques described in chapter 7.

9.9.3.3 Normal form of a relationship type

If relationship type R has no attributes, or if it has atomic and single-valued attributes only, R can be interpreted as a 1NF relation schema. The following example shows how relationship type ASSIGNED is given a relational interpretation.

```
rel-type ASSIGNED (    [0-N]: ORDER,
                       [0-N]: PRODUCT,
                       [0-N]: SUPPLIER,
                       QTY: integer)
```

can be interpreted as relation schema (note that the key has not been mentioned yet) :

```
ASSIGNED (ORDER,PRODUCT,SUPPLIER,QTY: integer)
```

This translation allows us to give E-R functional dependencies a simple relational interpretation as well :

```
rel-type ASSIGNED (   [0-N]: ORDER,
                      [0-N]: PRODUCT,
                      [0-N]: SUPPLIER,
                      QTY: integer)

PRODUCT --> SUPPLIER
```

can be interpreted as follows,

```
ASSIGNED (ORDER,PRODUCT,SUPPLIER,QTY: integer)
PRODUCT --> SUPPLIER
```

It is therefore quite valid to define the concept of normal form for relationship types. For instance, the ASSIGNED relationship type is in 1NF but not in 2NF.

9.9.3.4 Normalization of a relationship type

We shall limit our concern to situations where both the LHS and the RHS of the bad FD include an entity type, and where the LHS and RHS do not collect all the entity types. Other situations can be solved by first transforming the relationship type into an entity type (see chapter 7), or by using more general transformations proposed in [HAINAUT,91a]. The normalization process strictly follows the relational procedure. The example described above can be normalized by a decomposition based on the mentioned

```
rel-type ASSIGNED (   [0-N]:ORDER,
                      [0-N]:PRODUCT,
                      QTY:integer)
```

```
rel-type SUPPLIED (   [0-1]:PRODUCT,
                      [0-N]:SUPPLIER)
```

In general, some sort of inclusion constraint will be defined in reference with the semantics of the application domain. In the latter example, the following constraint seems the most natural since it asserts that we cannot assign a product that is not supplied by some supplier :

```
ASSIGNED[PRODUCT] is-in SUPPLIED[PRODUCT]
```

9.9.4 N1NF normal forms

9.9.4.1 N1NF theory : from Charybde to Scylla

The normalization principles presented so far are based on 1NF expressions of Entity-Relationship constructs. In particular, we have supposed that all entity type and relationship type attributes were atomic and single-valued. Analysing the properties of relation schemas with non-atomic and/or multivalued attributes makes us enter in the domain of non-first-normal-form (N1NF) relational theory. This theory is far more complex than the standard, 1NF theory (the simplicity of which can even be questioned!).

9.9.4.2 Practical analysis of N1NF data structures

Let's consider a first example (the specification of the domains has been dropped).

Example 1

```
entity-type CUSTOMER (
                CNUM,
                CNAME,
                ORDER [0-N] (
                          ONUM
                          ODATE)
                )
```

The description of a customer includes the description of his orders. Let's suppose that no two customers share the same order, and that all the orders have distinct ONUM values, whatever their customer. We can easily be convinced that this structure cannot induce any redundancies. There obviously exist more readable ways to express the same semantics, but this one cannot be criticized from the normalization point of view.

Let us try to study functional dependencies and identifiers that hold in this structure. CNUM is an identifier. In addition, we can easily derive the following FD :

```
CNUM --> CNAME
```  (since CNUM is the identifier of CUSTOMER)

`ONUM --> ODATE` (an order has one date only)

`ONUM --> CNUM` (since an ORDER value appear in one CUSTOMER entity only, and CNUM is an identifier of CUSTOMER)

We observe that ONUM determines (identifies) CNUM, CNAME (by transitivity), ODATE, and of course itself. Therefore, ONUM is another identifier of entity type CUSTOMER : given a value of ONUM, we cannot find more than one CUSTOMER entity that has this value.

Finally, we observe that each LHS is an identifier. Until now, applying 1NF reasonings on N1NF doesn't seem to work too badly.

The second example is as follows.

Example 2

```
entity-type ORDER(
              ONUM
              DATE
              LINE[1-N](
                        PNUM
                        PNAME
                        QUANTITY)
              )
```

The description of an order includes the description of its lines of order. We suppose that a product (through its PNUM value) may be mentioned only once in the LINE values of an ORDER entity. If we consider that the name (PNAME) of a product depends on its id-number (PNUM), then the name of a given product will be duplicated as many times as it appears in lines of orders. This structure will obviously induce a huge amount of redundancy and cannot be qualified normalized Let us analyse the FD's of this structure :

`ONUM --> DATE` (since ONUM is an identifier)

`PNUM --> PNAME` (since PNAME depends on PNUM only)

`ONUM,PNUM --> QUANTITY` (since no two lines may mention the same product)

This set of FD is somewhat more complex than that of example 1. In particular, the question of identifiers is not as obvious. It is clear that ONUM identifies CUSTOMER entities. In addition, let's consider all the LINE values occurring in all CUSTOMER entities. The same value may occur in more than one entity. Therefore, the set of LINE values has no intrinsic identifier. However, given a value of ONUM and a value of PNUM, we can point out one LINE value only (since a product cannot be mentioned more

than once in an order). We can then consider some sort of extended identifier for LINE values, namely (ONUM, PNUM)[7].

Pushing the reasoning a bit further, we can summarized the FD structure of LINE values as follows :

```
PNUM --> PNAME
ONUM,PNUM --> QUANTITY
id(LINE) : ONUM,PNUM
```

By analogy with 1NF relation schemas, we observe a situation that is somewhat similar to non-2NF relation schemas : an attribute (PNAME) depends on a proper subset only (PNUM) of the key (ONUM,PNUM). Furthermore, according to the BCNF criterium, there exists a FD the LHS of which is not a key. Thanks to a slight twist in the definition of the identifier of a multivalued, non-atomic attribute, we have succeeded in applying normalization criteria from the 1NF theory.

Let's now consider a third example.

Example 3

```
entity-type SUPPLIER(
              SNUM
              SNAME
              SALES[1-N](
                        PNUM
                        DATE
                        QUANTITY)
              )
```

The SUPPLIER entity type describes suppliers together with their past sales. For each sale of a supplier, we record the product number, the date and the quantity. Let's suppose that a supplier has not sold more than once a given product on a given date (at least from the point of view of the company which has been sold this product). We observe that this structure cannot imply any redundancies.

The FD holding in the set of attributes are as follows :

```
SNUM --> SNAME
SNUM,PNUM,DATE --> QUANTITY
```

In addition, SNUM is the identifier of entity type SUPPLIER, while (SNUM, PNUM, DATE) is the identifier of SALES values.

---

[7] See 2.3.10 for a more general discussion on attribute identifiers

Therefore, applying the normalization criteria used in example 2, we observe that each LHS is a key.

9.9.4.3 Normal form of N1NF data structures

We can then summarize the experiments that have been carried out in the previous section as follows.

1. Consider the attribute structure of entity type E as a tree, the root or level 1 of which is E. The direct attributes of E form level 2. If a level N attribute is multivalued and compound, then its component are at level N+1. If a level N attribute is compound but single-valued, its components can be considered at level N as well (i.e. we *flatten* these attributes).
2. For each level, we determine the identifiers. The identifier of an attribute at a level N greater than 1 can be made up of[8],

   a) either attributes from level N+1 (cfr example 1),

   b) or an identifier of level N-1 plus attributes from level N+1 (examples 2 and 3).
3. For each node in the tree that has components, we determine all the FD whose RHS is a non-key attribute.
4. If some FD have a LHS that is not an identifier of the upper node, then the entity type is not normalized.

Note that this procedure includes 1NF BCNF evaluation as well.

Let's apply these principles to the examples above.

Example 1

first step : build the attribute tree

```
CUSTOMER
    CNUM,
    CNAME
    ORDER
        ONUM
        ODATE
```

second step : determine the identifiers

```
CUSTOMER                id : CNUM
   CNUM,
```

8 More complex identifiers can be defined. They will be ignored in this manual.

```
   CNAME
   ORDER                id : ONUM
       ONUM
       ODATE
```

third step : determine the FD

```
CUSTOMER                id : CNUM
     CNUM,
     CNAME              CNUM --> CNAME
     ORDER              id : ONUM
          ONUM
          ODATE         ONUM --> ODATE
```

fourth step : evaluate the normalization state of CUSTOMER :

```
 all LHS are keys : the entity type is normalized.
```

Example 2

first step : build the attribute tree

```
ORDER
     ONUM
     DATE
     LINE
          PNUM
          PNAME
          QUANTITY
```

second step : determine the identifiers

```
ORDER                   id : ONUM
     ONUM
     DATE
     LINE               id : ONUM,PNUM
          PNUM
          PNAME
          QUANTITY
```

third step : determine the FD

```
ORDER                   id : ONUM
     ONUM
     DATE               ONUM --> DATE
     LINE               id : ONUM,PNUM
          PNUM
          PNAME         PNUM --> PNAME
```

```
            QUANTITY        ONUM,PNUM --> QUANTITY
```

fourth step : evaluate the normalization state of ORDER :

```
    a LHS is not a key : the entity type is not normalized.
```

Example 3

first step : build the attribute tree

```
SUPPLIER
     SNUM
     SNAME
     SALES
          PNUM
          DATE
          QUANTITY
```

second step : determine the identifiers

```
SUPPLIER                id : SNUM
    SNUM
    SNAME
    SALES          id : SNUM,PNUM,DATE
          PNUM
          DATE
          QUANTITY
```

third step : determine the FD

```
SUPPLIER                id : SNUM
     SNUM
     SNAME              SNUM --> SNAME
     SALES              id : SNUM,PNUM,DATE
          PNUM
          DATE
          QUANTITY      SNUM,PNUM,DATE-->QUANTITY
```

fourth step : evaluate the normalization state of SUPPLIER :

```
    all LHS are keys : the entity type is normalized.
```

NOTE

As already suggested, the principles developed so far do not cover all the theory of N1NF normal forms. In particular, the concept of identifier has been simplified. Furthermore, we have considered common, but restricted, forms of FD, in which the

LHS is a subset of the components + the identifier of the upper level. In all generality, the identifier of a node can comprise attributes from any level and the LHS of a FD can comprise attributes from any level as well. A more comprehensive theory would be much more difficult to master and would be definitely useless in our context.

9.9.4.4 Normalization of a N1NF entity type

Let's recall that normalizing a 1NF entity type consists in extracting the attributes of the LHS and RHS of a bad FD, defining a new entity type with these attributes, and finally defining a relationship type between these entity types. This procedure cannot be applied when the bad FD concerns attributes that are at a level greater than 2. The trick is to transform the attribute structure in such a way that the resulting entity type is in condition for 1NF normalization. We have to restructure the tree so that the bad FD is between level 2 attributes and/or related entity types only.

Let us process in this way example 2.

The initial schema is as follows :

```
entity-type   ORDER(
          ONUM
          DATE
          LINE[1-N](
               PNUM
               PNAME
               QUANTITY)
          )

PNUM --> PNAME
ONUM,PNUM --> QUANTITY
```

Since the bad FD (PNUM --> PNAME) is at level 3, the structure must be replaced by the following, in which this FD is at level 2. This result has been obtained by transforming attribute LINE into entity type LINE through transformation 7.7.1 (one-to-many version). We observe that a FD has been transformed into an identifier.

```
entity-type ORDER(
          ONUM
          DATE)

entity-type LINE(
          PNUM
          PNAME
          QUANTITY)
id(LINE) : ORDER,PNUM
PNUM --> PNAME

rel-type OL(
```

```
          [1-N]: ORDER,
          [1-1]: LINE)
```

This situation is that of an unnormalized, 1NF entity type, namely LINE. Indeed, it includes a FD the LHS of which is not an identifier. By applying the procedure proposed in section 9.9.3.2, we obtain :

```
entity-type ORDER(
          ONUM
          DATE)

entity-type PRODUCT(
          PNUM,
          PNAME)

entity-type LINE(
          QUANTITY)
id(LINE) : ORDER,PRODUCT

rel-type OL(
          [1-N]:ORDER,
          [1-1]:LINE)

rel-type PL(
          [0-N]:PRODUCT,
          [1-1]:LINE)
```

NOTE

Such a normalization procedure can lead to a situation where an entity type is completely *emptied* of its attributes. Let's consider, for instance, the following schema which is a simplified version of the previous one :

```
entity-type ORDER(
          ONUM
          DATE
          LINE[1-N](
               PNUM
               PNAME)
          )
PNUM --> PNAME
id(LINE) : ONUM,PNUM
```

Once again, this entity type is not normalized. Let's first transform it as follows :

```
entity-type ORDER(
          ONUM
          DATE)
```

```
entity-type LINE(
          PNUM
          PNAME)
id(LINE) : ORDER,PNUM
PNUM --> PNAME

rel-type OL(
          [1-N]:ORDER,
          [1-1]:LINE)
```

Then let us normalize LINE :

```
entity-type ORDER(
          ONUM
          DATE)

entity-type PRODUCT(
          PNUM,
          PNAME)

entity-type LINE
id(LINE) : ORDER,PRODUCT

rel-type OL(
          [1-N]: ORDER,
          [1-1]: LINE)

rel-type PL(
          [0-N]: PRODUCT,
          [1-1]: LINE)
```

This pattern suggests immediately the application of a reverse engineering transformation, namely transforming an entity type into a relationship type (see 7.2.1):

```
entity-type ORDER(
          ONUM
          DATE)

entity-type PRODUCT(
          PNUM,
          PNAME)

rel-type LINE(
          [1-N]:ORDER,
          [0-N]:PRODUCT)
```

9.9.4.5 Normalization of a N1NF relationship type

The Entity-Relationship on which the reverse engineering reasonings are based includes relationship type with single-valued roles, i.e. roles that can be taken by a single entity only. Therefore, roles cannot induce N1NF structures. However, relationship type attributes can be non-atomic and/or multivalued. This will occur rather rarely. In this case, we suggest to transform the relationship type into an entity type (see chapter 7) and to process the latter as proposed above.

9.10 OTHER KINDS OF REDUNDANCIES

Besides structural and unnormalization redundancies, which account for most of the problems encoutered in practice, there are numerous other kinds of technical structures that can be added in data structures in order to facilitate procedural parts of the applications. Let's mention two examples.

9.10.1 Counters

A counter is an attribute the value of which indicates the number of associated values or entities. In the following example, the attribute NUMBER-OF-LINES gives the number of LINE entities that depend from each ORDER. This attribute is redundant since its value can always be computed by counting the LINE entities.

```
entity-type ORDER(
          ONUM
          DATE,
          NUMBER-OF-LINES)

entity-type PRODUCT(
          PNUM,
          PNAME)

entity-type LINE
id(LINE) : ORDER,PRODUCT

rel-type OL(
          [1-N]:ORDER,
          [1-1]:LINE)

rel-type PL(
          [0-N]:PRODUCT,
          [1-1]:LINE)
```

9.10.2 State values

Some records in files may contain technical values instead of describing real world objects. Some examples :

- number of records in the file,
- highest current value of order numbers, customer numbers, etc,

9.10.3 Hierarchical level coding

A recursive data structure often represents trees or hierarchies of entities. Recording the level of each entity in the tree or in the hierarchy can be useful for the programs that navigates in these structures, specially when they are written in non-recursive languages. This information is clearly redundant since it can be computed by parsing the structure. In the example that follows, each PRODUCT entity is characterized by its level in the decomposition tree, and by the number of decomposition levels with which it is associated.

```
entity-typePRODUCT(
          PNUM
          PNAME
          LEVEL-NUMBER
          NUMBER-OF-LOW-LEVELS)

rel-type INCLUDES(
          high[0-N]:PRODUCT,
          low[0-1]:PRODUCT)
```

Chapter 10

THE MULTIPLE VIEW PROBLEM

Quite naturally, reverse engineering a database application leads to extracting several descriptions, or views, of the source database. These descriptions have to be merged, or integrated in order to obtain a single, global description. This chapter analyses the problems induced by the multiplicity of the data descriptions, and proposes techniques to derive a single schema from them.

10.1 INTRODUCTION

10.1.1 Motivations

The analysis of data-centered applications naturally leads to the production of several descriptions of the same data structures. Let us consider two representative examples.

1. In standard files applications, each program perceives the data according to its proper needs. In particular, some files are described while the others are ignored, some record descriptions are provided while the others are discarded, the relevant fields are described while the others are hidden. In other words, the program needs an interface to the data which is a partial view of them.
2. An application using a database has to access its content through a *subschema* or a *view*, which is made of the description of the data structures needed by the program, and that leaves aside the data structures that it does not need. CODASYL *subschemas*, IMS *PCB/PSB*, relational *views* are some examples of this concept.

Each of these partial descriptions is a view on the database structure. Considering that the ultimate aim of the reverse engineering process is the reconstruction of the conceptual

schema, it is obvious that such views are only input for this reconstruction. An important activity appears clearly, namely merging all the views obtained by source analysis, and producing a unique schema that encompasses the descriptions provided by these views. This merging process is generally known as **view integration**.

Here too, two situations will be distinguished :

1. In standard files applications, the views are the main source of knowledge on the actual data structures. No global DMS schema is available (except in some cases, where a data dictionary is used for instance). On the contrary, this global schema will be obtained through integration of these partial views.
2. In DBMS applications, a global DBMS schema is available, either as a DDL text, or as data dictionary contents. One could conclude that the views are no longer needed, and that the view integration process can be avoided. Unfortunately this is not true. Indeed, the global DBMS schema, as generally available, does not include all the conceptual specifications. Quite often, integrity constraints, field descriptions or explicit names have not been included in the global schema, but rather in the views. Consequently, view integration is still a basic process for this class of applications.

To be quite correct, view integration is an ubiquitous concept in data base engineering, and is by no means limited to reverse engineering. Requirements analysis, conceptual schema design, distributed databases, federated databases, are some major examples of situations and activities that need integrating data structures in order to produce a common view.

Due to the scope of this manual, we shall consider that being provided with several views of the database is basicaly a problem than we have to solve. This problem will be called **multiple view**, or **data description redundancy**.

This chapter presents view integration as a general problem related to database engineering. When required, more specific discussion will take place on the reverse engineering aspects.

10.1.2 Usual presentation of the multiple view problem

In the literature, the multiple view problem generally occurs in the conceptual design, as illustrated in Figure 10.1 :

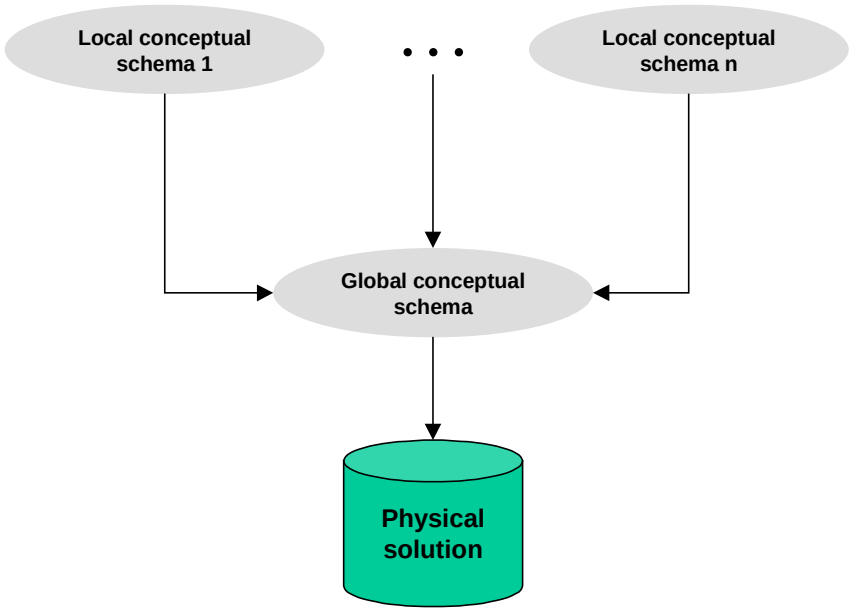


Figure 10.1 *- Usual framework of the conceptual schema integration problem : local conceptual schemas are integrated into a unique conceptual schema*

For large problems, a global view of the database to be implemented cannot be produced in one time. Local views are first produced, generally by different persons (allowing in this way the end-user participation to the specification process). Then a global conceptual schema is produced by merging these local views and can be implemented.

Several remarks have to be done about this process :

1. It is possible to find in this process elementary subparts which contain no redundancies. These are usually called "schemata". They correspond grossly to small part specified by only one person. So the term "schema" refers at the same time to a unit of work (of a person), and to an elementary non-redundant set of data structures. That explains also the usual "schema integration" expression. People using this expression always supposes there are no intra-schema redundancies.
2. Involved data structures are of a high abstract level, i.e. conceptual. This explains the use of terms like facts, real-world (state), ... and more generally a high level referential (= what does a data structure correspond to?), used for similarity detection.
3. The integration process is vital here because it is out of the question that the following step - the implementation - will work on data structures including redundancies (see chapter 9.1.1 for the drawbacks of the redundancy). That explains why the problem of multiple view was originally studied in this framework.

10.1.3 A more general approach of the multiple-view problem

The usual approach of the problem is not sufficient. More particularly, it doesn't tackle exactly the problems encountered in the database reverse engineering. The three points of the previous section can in fact be negated in the case of reverse engineering :

1. It is not always possible to find elementary non-redundant data structures in the reverse engineering process. The smallest physical data structure description can be redundant with another one.

   So the usual significance of the term "schema", as a set of non-redundant structures, does not hold in this case (the notion of schema remains useful, but only in the sense of unit of work; this concept is user-controlled).

2. The problem of redundancy is, at least for poor D(B)MS, more important at the physical level (analyzing the physical data description redundancies) than at the conceptual one. So physical concepts must be managed during this type of integration. Special attention must be paid to physical storing of data, that means the way data are implemented : length, position, type, ... Physical data structure integration uses also other similarity detections, than the real-world referential of the conceptual l

3. The integration process is less important than for the conceptual design. It remains essential for most people, interested by a global view of the physical database, but other possible persons, for instance the DBA, could be interested by the different views of the physical database only.

The solution in this case consists in not eliminating the original views during the integration, allowing by this way several co-existing versions of the data structures (see chapter 13).

Our intention in the following is therefore to present a more general approach of the multiple-view problem. We will tackle the different possible levels of description of data structures. We will also take up the problem of the diversity of representations of a same fact in the E/R model (this problem was mentioned in chapter 7, in the transformational approach). In fact - and this explains also why the integration problem was studied by so many well-known researchers in E/R -, trying to solve this pro

10.2 EXAMPLES

The following sections will define the theoretical principles of integration. These principles could be sometimes difficult to understand. Therefore, we will start with an example, which will be analyzed all along this section.

Let us suppose we have a Cobol program, which works on the order management of a company. There are two 01 record types corresponding to the logical file `FIL-ORD`.

```
program-id P1.
...
input-output section.
file-control.
     select FIL-ORD assign to DSK:FIL-00345
     record key is ORD-NUM of ORDER1.
...
fd FIL-ORD.
01 ORDER1.
     02 ORD-NUM         pic X(8).
     02 ORD-DATES       occurs 2 times pic X(6).
     02 CUS-NUM         pic X(7).
     02 CUS-STREET      pic X(15).
     02 CUS-NB     pic X(4).
     02 CUS-ZIP         pic X(4).
     02 CUS-CITY        pic X(20).

01 ORDER2.
     02 ORD-NUM         pic X(8).
     02 ORD-DATE-CUS    pic X(6).
     02 ORD-DATE-FUR    pic X(6).
     02 filler          pic X(7).
     02 CUS-ADDR        pic X(42).
```

From this program, we can extract two record types, which are two views of the same logical file. Suppose we extract also a third (more conceptual) view, from an interview with the ordering manager. These three views are illustrated in figure 10.2.

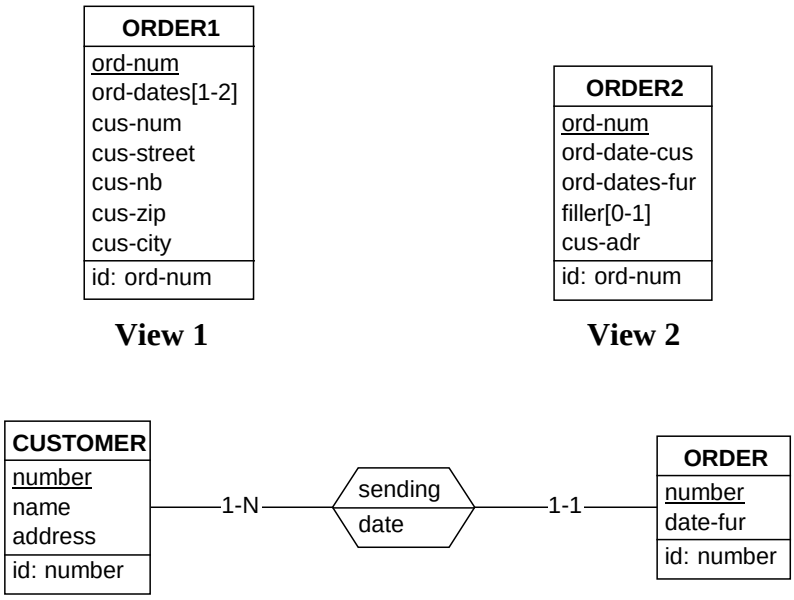


Figure 10.2 *- Basic example for the chapter*

These three views will be our basic illustrations for the following sections.

10.3 GENERAL INTEGRATION PROCESS

If we want to realize more than a simple juxtaposition of these three views, i.e. a real merging, our problem can be divided into two steps :

- the first one consists in detecting and asserting what is common, i.e. the semantic relationships or correspondences, between the different structures of the views. The correspondences are the positive results of this step; negative ones are the conflicts, which are assertions of incompatibility between two structures.

  This step cannot be fully automated, but only assisted, because of the semantic estimations it requires (it is therefore a major work of the integration).

  Example of correspondence : the ORDER1s in view 1 "are the same objects" than the ORDERs in view 3.

- the second one consists in integrating the different views, according to the correspondences and conflicts generated in the first step. This can be processed (almost) automatically.

  Example : according to the previously defined correspondence, only one entity type ORDER will be produced by the integration process in the resulting structure.

Note : we do not assume a total sequentiality of these two steps :

- either all the correspondences between two views are asserted before the integration is performed,
- either these two processes are carried out in parallel.

10.4 GENERAL STRATEGIES

Generally speaking, more than two structures can have redundancies (as in our example). The following strategies are possible and presented in [Batini86] :

- direct (or one shot) integration of the n structures, whatever the value of n.

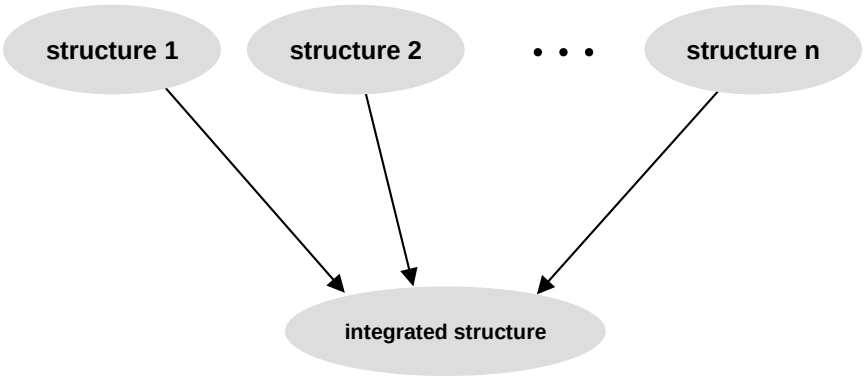


Figure 10.3 *- N-ary integration strategy*

- integration of a pair of structures; then the result is integrated with the integration of another pair, if any (if not, an original structure is used). This strategy is called balanced treelike strategy.

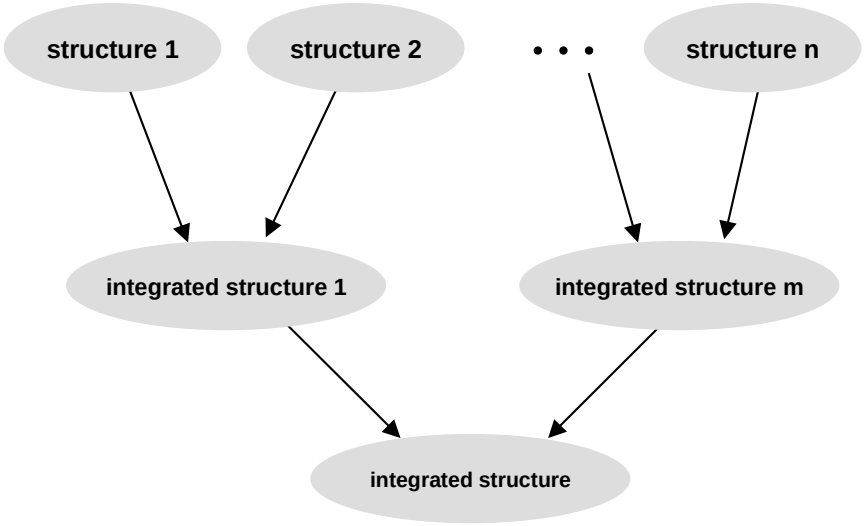


Figure 10.4 *- Binary tree-like integration strategy*

- integration of a pair of structures; then the result is integrated to another original structure (not balanced treelike or incremental strategy). A same new structure is in fact enriched by successive integrations.

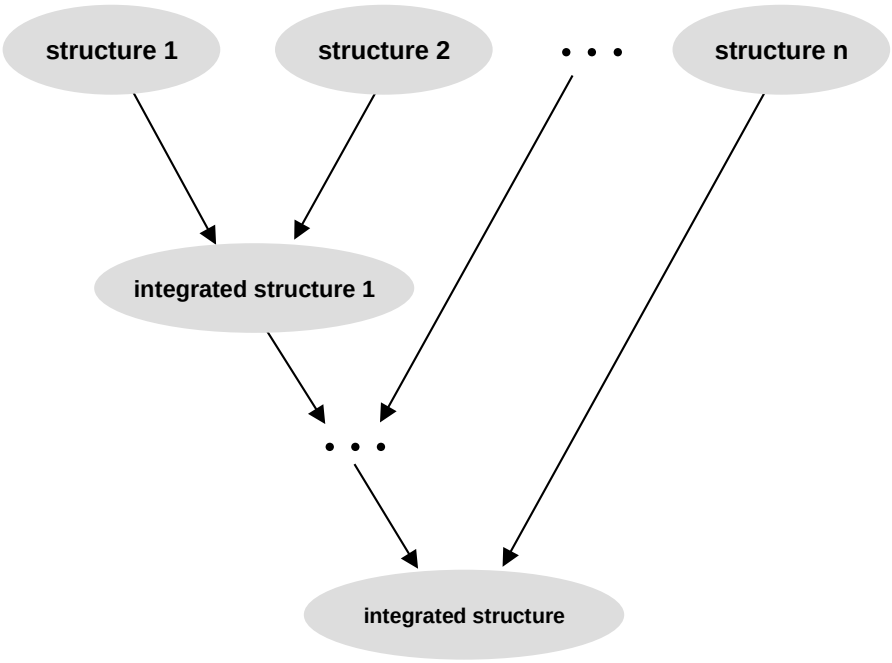


Figure 10.5 *- Binary incremental integration strategy ("ladder" approach)*

- mixing the three previous strategies.

Because of the complexity of the N-ary strategy, both for the theoretician and for the user, we will adopt a binary strategy. We will explain in the following the integration process of two data structures, which could be usable either in a balanced treelike strategy, or in an incremental one. In particular, that means that the concept of correspondence that will be developed in the next section, is a binary relationship.

For our example, a good strategy would be to integrate first the views 1 and 2, then integrating the result with view 3.

10.5 A SEMANTIC NETWORK VIEW OF THE E/R MODEL

The transformational approach developed in chapter 7 has shown that a same structure can be transformed into several equivalent ones. That means that the same semantics can be represented in several ways. This makes the integration more difficult. We need a more generic approach of the E/R model which will allow us to express more fundamental semantic correspondences.

The approach we propose here consists to consider an entity-relationship schema as a set of classes and paths.

10.5.1 Classes

Classes (entity types, relationship types, attributes) are the nodes of the semantic network. Each class has a name (except for special cases, such as the FILLER clause in Cobol).

Example : in view 3, CUSTOMER, SENDING and address are classes.

Note that this approach is strongly influenced by the object-oriented approach. As a consequence, attributes will be considered independently of the entity/relationship types (therefore, we will sometimes draw them independently of their father, linking them by a line). Sometimes, it will be difficult to apply this approach to physical data structures, which are more value-oriented.

10.5.2 Arcs

There are two kinds of arcs :

- the roles, and
- the relationships between an attribute and its father.

An arc has sometimes a name. We will note arcs simply by the linked classes, related by a "$\times$".

Example : in view 3, "sends" (CUSTOMER $\times$ SENDING) and the link number $\times$ CUSTOMER are arcs.

An arc has also cardinalities, in both directions, which express the minimum and maximum number of objects which must or could be linked by the arc. More precisely, they corresponds to already defined numbers :

- for an arc of type "attribute-father", its cardinalities in the direction father $\varnothing$ attribute are the repeating factors of the attribute; in the direction attribute $\varnothing$ father, the minimum cardinality is 1 (because an attribute value is always linked to its father) and the maximum one is either 1, either N, according to the fact the attribute is an identifier of its father or not.

  Example : the arc number´CUSTOMER is 1-1 in both directions.

- for an arc of type "role", its cardinalities in the direction relationship type $\varnothing$ entity type are 1-1 (because of the definition of the relationship type). They correspond, in the direction entity type $\rightarrow$ relationship type, to the cardinalities of the role (as defined in chapter 2).

  Example : the arc "sends" is 1-N in the direction CUSTOMER $\rightarrow$ SENDING and 1-1 in the direction SENDING $\rightarrow$ CUSTOMER.

10.5.3 Paths

A path is a list of arcs and intermediary classes, linking two classes.

$P = arc_1 \times class_1 \times ... \times arc_n \times class_n \times arc_{n+1}$ $(n \geq 0)$.

This concept of path is important. It allows us to define and compare structures which are at different levels of detail or aggregation. We will note paths by enumerating the two linked classes, and the intermediate ones, separated by " × ". Note that an arc is also a path (when n = 0).

Example : the path ORDER × SENDING × CUSTOMER × address relates ORDER and address.

The definition of arc cardinalities can be extended for paths. The cardinality values of a (non-cyclic) path P can be computed from the cardinalities of the arcs arc_1, arc_2, .., arc_k is made up of :

$$\text{min-card}(P) = \text{min-card}(arc_1) * \text{min-card}(arc_2) \ldots * \text{min-card}(arc_k)$$

$$\text{max-card}(P) = \text{max-card}(arc_1) * \text{max-card}(arc_2) \ldots * \text{max-card}(arc_k)$$

We will sometimes use the term "concept", as a generalization of classes and paths.

Using this more generic view will allow us to define more general integration processes, defining for instance semantic correspondences between classes of different types, or between structures of different levels of aggregation.

10.6 SEMANTIC CORRESPONDENCE

10.6.1 The concept of real-world facts

The process of integration is in fact a union process, that means collecting objects which are different but keeping one instance of each only.

The problem is to define what "two object instances are identical" means. An interesting definition is proposed in [Larson89] and [Spaccapietra 90], which define the concept of real-world state of an object class, i.e. the set of real-world facts represented by instances of this class. So comparisons between two views will use this concept of real-world state. For instance, entity types will be declared as equivalent or identical if they have the same real-world state. Let us take two examples of real-world facts :

- the most obvious case is a conceptual schema : the entity type CUSTOMER in view 3 refers to the actual persons or firms which are customers. Their address refer to the real addresses used by the post office.
- for physical schemas in the reverse engineering context, schema constructs are considered as representations of data structures. The real-world facts represented by these structures are the data stored in a database. For instance, the real-world facts represented by the record type ORDER1 in view 1 are the data records recording real world orders.

Correspondences can be stated between physical structures according to the correspondences between the data sets described by these structures.

10.6.2 Semantic type of correspondence

Our intention in this section and in the next one is to systematize the different possibilities which can be encountered in the multiple-view problem. This section will give a first dimension to organize the different eventualities, according to the semantic type of correspondence. The three following semantic correspondences can be distinguished between structures (the word "structure" will be refined in the next point) :

Case 1 : a structure is identical to another structure

The first obvious type of correspondence. It is an equality or a bijection of the real-world facts described by these two structures. Note that the concept of bijection will allow us, for instance, to define correspondences between physical-oriented views and more semantic ones (the interested reader can find in [Spaccapietra90] more precise mathematical definitions).

The correspondences we will assert for our example will belong to this case (for instance, identity of ORDER1 in view 1 and ORDER2 in view 2).

Case 2 : a structure is a generalization of another structure

A structure A is a generalization of a structure B if each real-world fact b described by B is also described by A, i.e. if the set of the real-world facts of B is included in the set of the real-world facts (or a bijection thereof) of A.

Example :The entity type PERSON is a generalization of the entity type EMPLOYEE.

The solution consists in expressing both structures, and in explicitating the inclusion relationship between these two structures. This inclusion relationship could later be replaced by other structures, using the transformations developed in chapter 7.

This case will not be studied in the following.

Case 3 : Two structures are specializations of the same unknown structure

Example : The entity types MAN and WOMAN.

This more complicated case can be resolved by using the previous cases : Let A and B be these two structures; let C be the unknown more generic structure. C must be expressed;

then A and B have to be integrated using the technique of case 2. Note that C can be expressed, either as the union of A and B, or as a wider concept.

This case will not be studied in the following.

10.6.3 Number of involved concepts

A second dimension to classify structures which correspondences can be defined on, is to consider the number of involved concepts.

The simple case is the correspondence between two concepts : a concept corresponds (using the different cases of the previous point) to another concept.

But it happens frequently that correspondences can be defined between several concepts and one or several concepts. More precisely, it is as if such a correspondence was asserted on an *implicit* concept, which groups several ones. The attribute cus-adr in view 2 illustrates this situation : its corresponding structure in view 1 is made up of cus-street, cus-nb, cus-zip and cus-city.

There are several ways to group several concepts into a single one. We are going to distinguish three of them.

10.6.3.1 Class aggregation

This process was used in the above example. It is often used in physical data model; therefore, it will be helpful to integrate physical structures.

We will study particularly this process for fields (attributes), when one field in a view corresponds to the aggregation of several fields in another view.

10.6.3.2 Arc composition

Our definition of path expresses the fact that a path can be seen as a composition of arcs. Since we have given an explicit concept to this composition, we will be able to define correspondences between a concept and a path, i.e. several arcs.

Example : in our example, the arc ORDER×ord-date-cus in view 2 corresponds to the composition of the arcs ORDER×SENDING and SENDING×date in view 3.

10.6.3.3 Grouping

Another important abstraction process is the grouping : a structure which represents a list (or a set) of facts can correspond to several structures representing these facts individually.

Example : the field ord-dates in view 1 is the grouping of ord-date-cus and ord-date-fur in view 2.

This process can be applied not only to classes but also to paths, as it will be detailed in next section.

Note : more complicated situations are possible if some (but not all) concepts exist and have to be put in correspondence with the aggregation or grouping concept (for instance, there are street and number in a view and address in a second view).

We will not tackle this cases in the following

10.6.4 Comparable structures

In the two previous sections, we have just defined two ways to classify semantic correspondences. By combining these two dimensions, here are the list of comparable structures, that we will consider.

10.6.4.1 Class/Class correspondence

A class corresponds to another class. In simplest cases, an entity type can correspond to an other entity type. But it can also correspond to a relationship type or an attribute (see the transformations of an entity type into a relationship type or in an attribute, and the reverse ones).

Example : the entity type ORDER1 in view 1 corresponds to the entity type ORDER in view 3.

Six combinations between these different types can occur :

- entity type/entity type
- entity type/attribute
- entity type/relationship type
- relationship type/relationship type
- relationship type/attribute
- attribute/attribute

10.6.4.2 Class/Classes correspondence

We will retain only the following situations :

- an attribute corresponds to the aggregation of several attributes.
- a multivalued attribute corresponds to several attributes.

10.6.4.3 Path/Path correspondence

We can analyse three specific cases.

Arc/Arc correspondence

This is the simplest case. This case does not induce special problems.

Example : the arc ORDER1×ord-num in view 1 and ORDER2×ord-num in view 2 are equivalent.

Arc/Multi-arc path correspondence

An arc can correspond to a path. That means that the arc represents structures which are more concise than the corresponding structures of the path.

Examples :

- the arc ORDER×ord-date-cus in view 2 corresponds to the path ORDER×SENDING ×date in view 3.
- the arc ORDER×cus-num of view 1 corresponds in view 3 to the path ORDER×SENDING×CUSTOMER×number.

This situation comes from one of the main ambivalence of the E/R model, illustrated, for instance, by the transformation of a relationship type into an entity type.

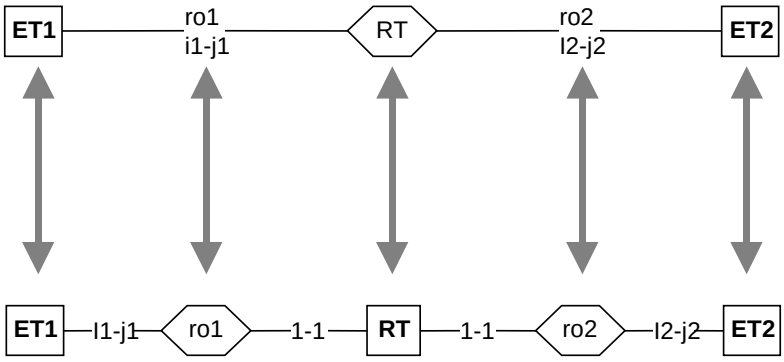


Figure 10.6 *- The correspondence between the relationship type RT and the entity type RT implies correspondences between the arcs ro_1 , ro_2 and the classes ro_1 , ro_2 .*

This ambiguity consists of the fact that a mathematical relationship between two sets can be semantically interpreted in two ways (see [Codd79]) :

- either strictly as a subset of the cartesian product of the two sets. The relationship is not a concept by itself. This notion is close to access paths, or 1-N relationship types without attributes.

  Example : the relationship type INCLUSION between the entity types SERVICE and DEPARTMENT.

- either as an intermediary class which relates two other classes. That class has instances, representing individual links. This perception can be found in usual relationship types, particularly for N-N, with attributes or n-ary (n>2) relationship types.

  Example : the relationship type LICENCE between PERSON and VEHICLE.

This ambivalence has to be (and will be) managed by the integration process.

Multi-arc path/Multi-arc path correspondence

A path corresponds to a path (and no more elementary correspondences of their components can be stated). This could occur in sophisticated and infrequent situations where the intermediary concepts of the paths are different.

10.6.4.4 Path/Paths correspondence

This case can occur if one path is the grouping of several paths.

Example (from [Spaccapietra,90]) :

[JLH-2012: sorry, this example is wrong. Must be revised]

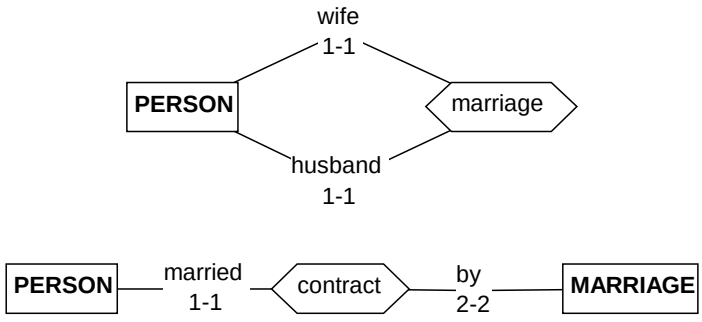


Figure 10.7 *- Example of a grouping path*

The path PERSON×CONTRACT×MARRIAGE in view 2 is a grouping of the arcs husband, wife in view 1.

10.6.5 How to detect equivalent structures

In this section, we will present some hints for correspondence detections. We will use the predefined classification.

10.6.5.1 Class/Class equivalence

Name comparison

Name correspondence is important for the detection of class equivalence. When two classes have names which are semantically synonyms (identical, substring or similar, for instance), it is a valuable hint of the equality of the two classes.

Example :cus-num (view1) and number (view 2) are equivalent attributes.

This rule holds particulary for relationship type or entity type equivalence assertions. When two attributes are involved, this rule is less significant.

For more details about name analysis, see chapter 6.

Length and format comparison

The fact that two classes have the same length (and the same format), if these informations are available, is also an important hint to detect class equivalences, because of the required length and format equalities we will define hereafter.

The coupling of this rule with the previous and/or next point strengthens strongly the probability of class equivalence assertions.

Context comparison

The problem of context comparison is more complex. The principle is to detect class equivalence by looking at the correspondence of the nearest structures. For instance, knowing that two entity types are equivalent is very helpful to assert that two of their attributes are equivalent. In our example, knowing that ORDER in view 1 corresponds to ORDER in view 2 is useful to reduce the search space for a correspondent of ord-num to the attributes of ORDER in view 2.

The problem is that, in a graph, it is a cycling reasoning. The example above can be inverted : knowing that two (identifying) attributes are equivalent is useful to establish an equivalence between two entity types ([Larson89]). So each structure correspondence is a hint to detect a correspondence of a close one, and conversely.

Generally, a correspondent of a structure must be searched in the closest structures of the correspondent of a neighbour structure.

More specifically, for an entity type, father(s) of the possible corresponding classes of its attributes and neighbours of the possible corresponding classes of its relationships have to be analyzed.

For an attribute, attributes(s) of the possible corresponding class of its father must be examined. If the information is available, the attribute which has the same address (position) in this class must particularly be examined.

Special case for physical data structures

For physical data structures, another important hint is the data memory space described by these structures. If two structures describe (or can describe) the same memory space, data that can be contained into can be described by these two structures. Therefore, these structures can be integrated. Let us take some examples :

- when, in Cobol, two record types belong to the same logical (or physical) file, they are in fact two buffer descriptions of this file. Sometimes, records contained in the file can be accessed by several record types, which may therefore be integrated.
- when there is a transfer instruction (assignement, for instance) between two variables, their description may be integrated.
- the REDEFINES clause in Cobol (or EQUIVALENCE clause in Fortran) asserts explicitly an equality of memory space between two data.

Note that, while this approach seems to be more automatic, there remain uncertainty factors : for instance, if a file is described by several record types, that could be due to the fact that it records several different clustered informations (the customers and their orders, for instance).

10.6.5.2 Class/Classes aggregation equivalence

Hints which allow to detect an equivalence between one class and several ones, are the same as those which help to detect an equivalence between two classes (because the aggregation/grouping of these several classes can be defined as a class).

For an aggregation attribute and several ones :

- the length (and format) of an attribute is (are) the same as the length (and format) of the concatenation of several attributes.
- the name of an attribute is similar to some common parts (usually prefix or suffix) of the names of several attributes.
- for the context, see the rules defined below for class equivalence detection.

10.6.5.3 Class/Classes grouping equivalence

The same reflexion holds for the correspondence between a grouping attribute and several ones :

- the length (and format) of an attribute is (are) the same as the length (and format) of the concatenation of several attributes (usually, these attributes have the same length and format).
- the name of an attribute is similar to some common parts of the names of several attributes (sometimes, the name convention is a numbering suffix added to a common part).
- for the context, see the rules defined below for class equivalence detection.

10.6.5.4 Path/Path equivalence

Trying to detect an equivalence involving an arc (path) is easy if the classes linked by the arcs have already be defined. The list of all the arc/paths linking the corresponding classes of the original classes have to be searched for a corresponding arc/paths. According to the preconditions we will define hereafter, names and cardinalities have to be analyzed to find an exact equivalent corresponding arc/path.

10.6.5.5 Path/Paths grouping equivalence

As for attribute/attributes grouping equivalence, we can refer to the previous simple case :

- a path link the same classes as other paths.
- a path has a name similar to some common parts of the names of several paths.
- a path has cardinalities which are equal to the sum of the cardinalities of several paths.

10.6.6 Example

As a conclusion for semantic correspondence, let us complete our preliminary example. The following correspondences can be defined (we have not retained the immediate arc/arc equivalences for attributes).

View1/View2

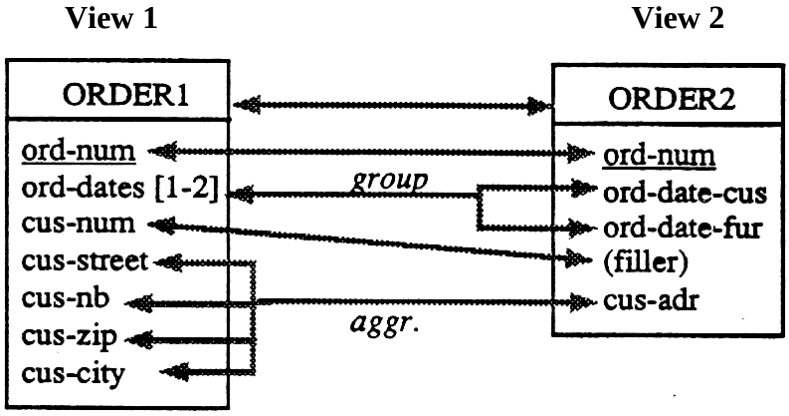


Figure 10.8 *- Basic example: class/class(es) correspondences (view 1/view 2)*

- class equivalences (view 1 / view 2) :
  - ORDER1 and ORDER2
  - ORDER1.ord-num and ORDER2.ord-num
  - ORDER1.ord-dates and (grouping of) ord-date-cus, ord-date-fur
  - ORDER1.cus-num and ORDER2.filler
  - (aggregation of) ORDER1.cus-street, ORDER1.cus-nb, ORDER1.cus-zip, ORDER1.cus-city and ORDER2.cus-adr

View1/View3

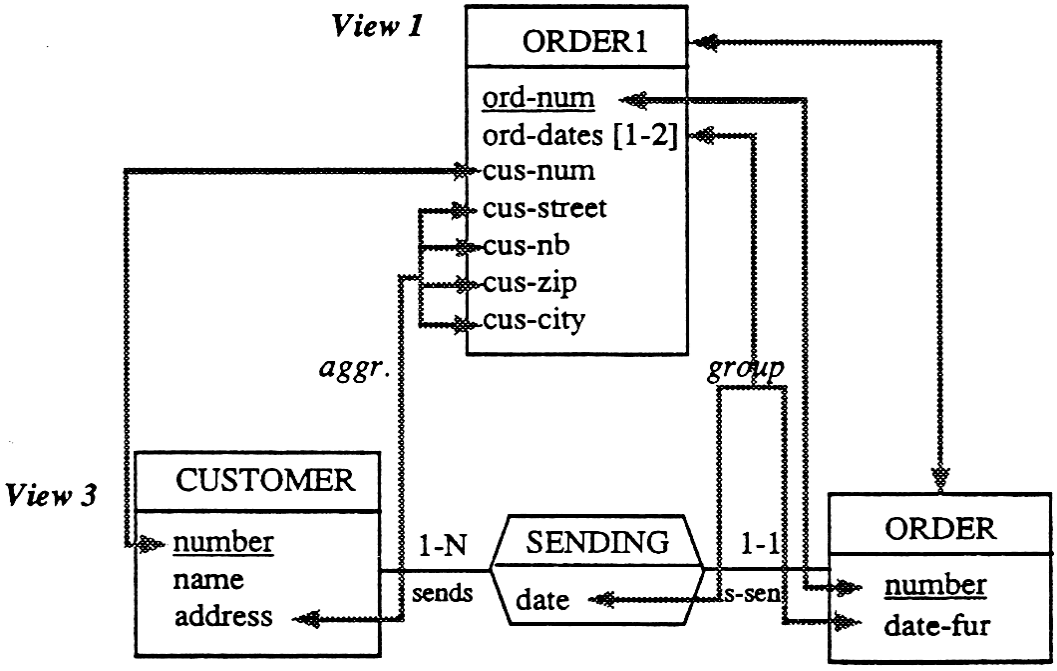


Figure 10.9 *- Basic example: class/class(es) correspondences (view 1/view 3)*

- class equivalences (view 1 / view 3) :
  - ORDER1 and ORDER
  - ORDER1.ord-num and ORDER.number
  - ORDER1.ord-dates and (aggegation of) ORDER.date-fur, SENDING.date
  - ORDER1.cus-num and CUSTOMER.number
  - (aggregation of) ORDER1.cus-street, ORDER1.cus-nb, ORDER1.cus-zip, ORDER1.cus-city and CUSTOMER.address

- path equivalences (view 1 / view 3):
  - ORDER1×ord-num and ORDER×number
  - ORDER1×ord-dates and ORDER×SENDING×date

    ORDER1×ord-dates and ORDER×date-fur
  - ORDER1×cus-num and ORDER×SENDING×CUSTOMER×number
  - ORDER×cus-street and ORDER×SENDING×CUSTOMER×address

    ORDER×cus-nb and ORDER×SENDING×CUSTOMER×address

    ORDER×cus-zip and ORDER×SENDING×CUSTOMER×address

    ORDER×cus-city and ORDER×SENDING×CUSTOMER×address

10.7 INTEGRATION OF EQUIVALENT STRUCTURES

After a brief overview of the global integration principles, this section will present in detail the different cases of integration. Then, our example will be completed.

10.7.1 Pre-integration step

This step, usually mentionned in the litterature, consists in preparing the views to be integrated, so that the integration process will be made easier. It consists generally in restructuring these views according to a given standard.

Because our intention is to propose a very generic description of the process, we hope this phase of pre-integration should be used rarely. However, because of the (almost) infinite diversity of the possible structures to be integrated, this process will sometimes consist in transforming (see chapter 7) some structures in such a way that they will be more comparable.

10.7.2 Principles of integration

Our basic integration principle is simple : every class and arc of both views must be present in the resulting view, whatever its form.

For structures which are present in only one view, they have to be simply present in their original form in the resulting view.

For equivalent structures, only one must be retained, with the most adapted representation, often the more detailed one.

10.7.2.1 Integration of two equivalent classes

Entity types, relationship types and attributes are classes. The result of their integration will be simply the class, in one of its original forms.

The main characteristics of a class are : a name, and at the physical level, a length and a format.

These characteristics (and also others, such as constraints, comments, ...) have to be managed during the integration. For instance, conflicts must be solved :

- if the names are different, one of them must be chosen (or a third one proposed)
- if the lengths are different, at the physical level, there is a main conflict : the real-world facts described by these classes (i.e. the data) are different. This conflict can be solved by choosing one length, but it is clear that all following reasoning processes using length calculations will be corrupted.
- if the formats are different, one of them must be chosen.

Note :It is important to recall that a class, together with other structures, can be put in correspondence in a whole path with another structure.

10.7.2.2 Integration of two equivalent paths

Two paths can be equivalent if they link classes which are themselves related by an equivalence correspondence. So having a resulting integration for each class of a path is necessary to be able to integrate this path itself.

Therefore, we will present the arc integration in the framework of the integration of the classes they relate : a relationship type will be integrated together with its roles, an attribute with the arc with its father.

Integrating paths means managing :

- their possible names
- the cardinalities

- their possible intermediary classes

Now, we are going to present, in a more specific way, these global principles : we will start with the simple case when a structure has no correspondent, then we will present the integration of the different structures correspondences defined in point 10.6.4. However, it will be often a simple application of these basic principles.

10.7.3 No correspondence case

10.7.3.1 No correspondent for an entity type

If an entity type in a view has no correspondent in the other one, it has simply to be added to the result.

10.7.3.2 No correspondent for a relationship type

If a relationship type has no correspondent, that means also its roles have no correspondent.

Note :We don't consider here the special case of path/path integration, which refutes this assertion (for instance, if a relationship type with its two roles are considered globally as a correspondent of a path).

The solution consists in creating a new relationship type, with new roles which are equivalent to the original ones (so they must be defined on the same entity type(s), known because of the precedence principle defined in 10.7.2.2).

10.7.3.3 No correspondent for an attribute

For an attribute without correspondent, an equivalent new attribute must simply be created to the resulting integration of its father (known, because of 10.7.2.2).

10.7.3.4 No correspondent for an arc

However, it is possible for the two kinds of arcs we have just considered, that their classes have a correspondent but the arcs themselves have not. For instance, we could have :

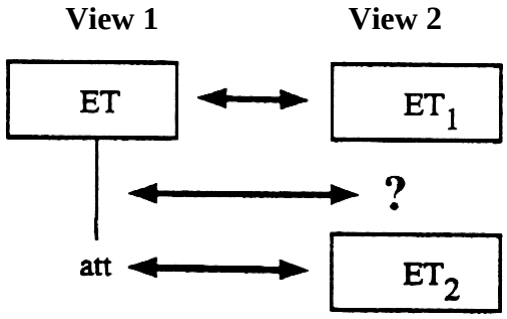


Figure 10.10 *- An arc without correspondent*

and no paths relating ET1 and ET2 can be found.

The solution consists in creating a "semantic" link between the classes to be related, with the characteristics of the arc, according to the types of these classes. This logic could be summarized by this small algorithm :

1. if two entity types are involved, the link to be created will be a relationship type.
2. if an attribute is involved, transform it into an entity type (see chapter 7 for the transformation) then re-consider the situation (goto 1).
3. if a relationship type and an entity type are involved, lpqif a role which respects the characteristics of the original arc can be created, then create this role between them; otherwise transform the relationship type into an entity type and re-consider the situation (goto 1).
4. if two relationship types are involved, transform one of them[1] into an entity type and re-consider the situation (goto 1).

10.7.4 Class/Class integration

We will develop the six possibilities defined in 10.6.4.1. The relationship type/attribute equivalence case will be given two interpretations. Let us recall we will consider simultaneously the integration of a class and of its environments.

[1] which one? in priority, the one which makes possible the creation of a role respecting the arc characteristics, linking the resulting entity type and the remaining relationship type.

10.7.4.1 Entity type/Entity type integration

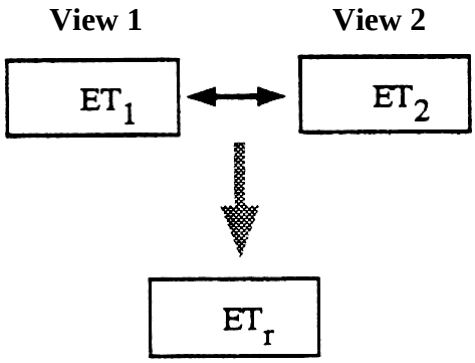


Figure 10.11 *- Entity type/Entity type integration*

Two equivalents entity types are integrated into a new entity type. Let us recall we are in an object-oriented interpretation, so attributes of these entity types are considered independently.

This case is the simplest one, because no paths are involved.

10.7.4.2 Entity type/ Relationship type integration

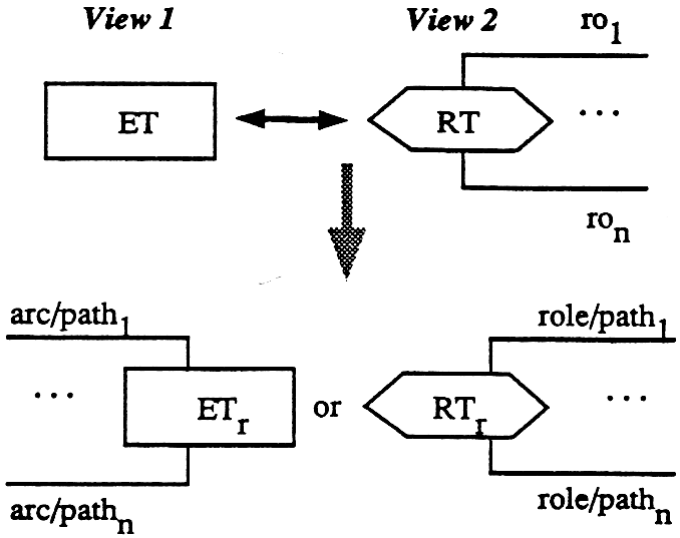


Figure 10.12 *- Entity type/Relationship type integration*

Integrating ET and RT means integrating also the arcs ro_1 , ..., ro_n (therefore, it is necessary to know the correspondents of the entity types of RT).

The resulting class should be a relationship type, if it is possible to express the resulting integration of the roles without be forced to transform RT into an entity type.

10.7.4.3 Entity type/Attribute integration

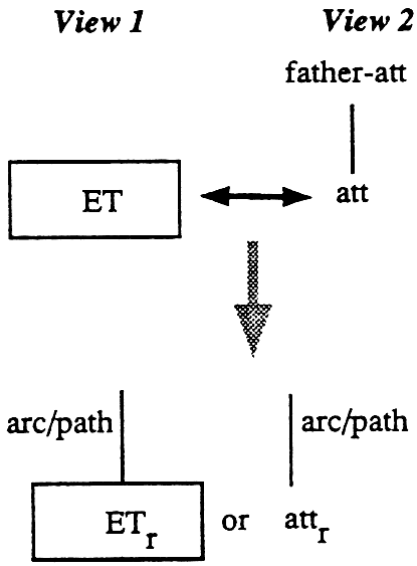


Figure 10.13 *- Entity type/Attribute integration*

The class represented by ET and att is represented only once in the resulting structure. The arc att×father-att has to be integrated, too. This integration could force the representation of the resulting class to be an entity type.

10.7.4.4 Relationship type/Relationship type integration

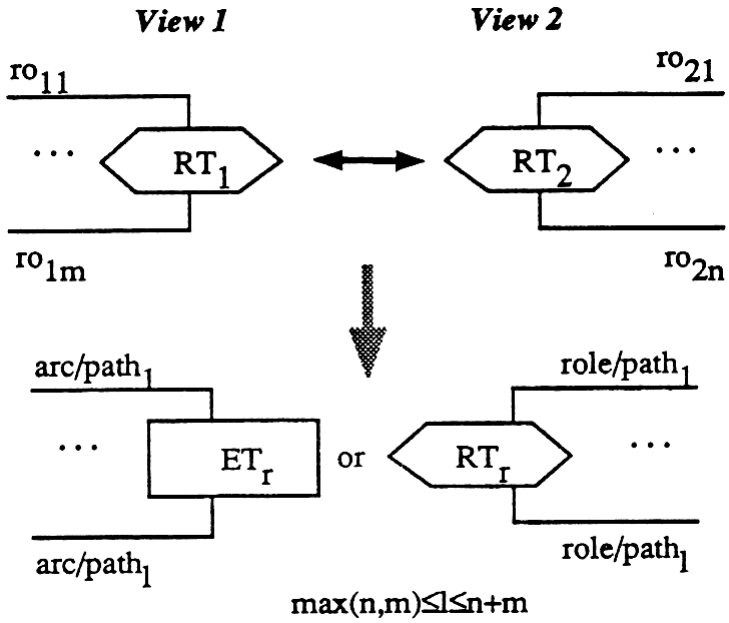


Figure 10.14 *- Relationship type/Relationship type integration*

We have here to integrate at the same time RT1 and RT2, plus the m roles of RT1 and the n roles of RT2. So the result will be an entity type if it is required by these integrations.

10.7.4.5 Relationship type/Attribute integration

Two cases have to be considered here :

- the first one tackles the usual case of class equivalence,
- the second one tackles the special case where an attribute and a referential integrity constraint correspond to a relationship type.

Relationship type/Attribute class integration

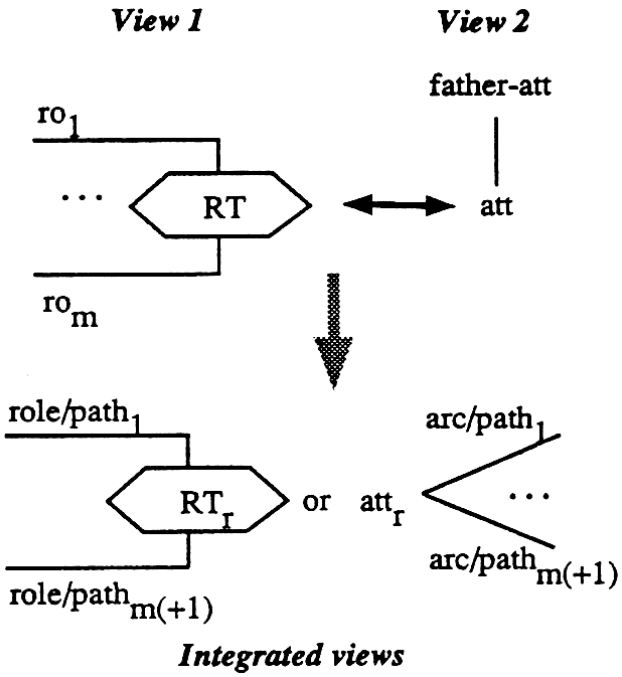


Figure 10.15 *- Relationship type/Attribute integration : first case*

The case requires the integration of RT and att, and also the integration of ro1, .., rom and of the arc att×father-att[2].

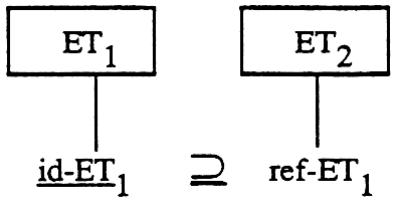
---

2 Note there is a possible deadlock if an entity type E of a roi is equivalent to a subattribute A of att, because the integration of RT require to know the correspondent A of E, and E cannot be integrated with A if att is not integrated with RT.

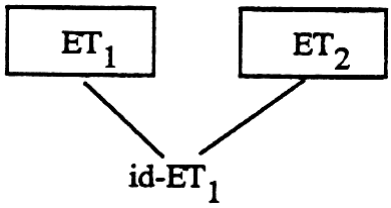
Relationship type/Referential attribute integration

What is a referential integrity constraint? It indicates that a constraint of inclusion (or even equality) exists between two (sets of) attributes to record a semantical link. This is an implicit way to record a path.

The pattern :



could be redrawed as :



which explicits the path.

So there are cases when a relationship type between ET_1 and ET_2 will have to be integrated with a referential attribute.

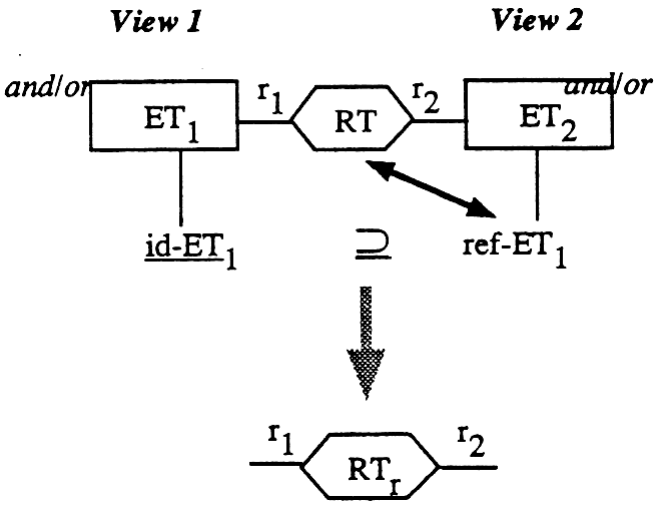


Figure 10.16 - *Relationship type/Attribute correspondence : second case*

The result of this integration will be the relationship type (because it is a more explicit structure).

10.7.4.6 Attribute/Attribute integration

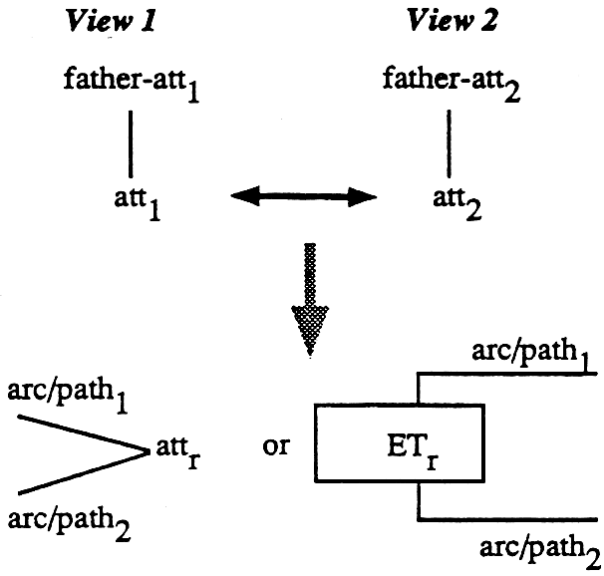


Figure 10.17 *- Attribute/Attribute integration*

If two attributes are equivalent (this case must not be confused with the previous one), the result of their integration will be an attribute or, if the integration of their arcs with their fathers requires it, an entity type.

10.7.5 Class/Classes integration

10.7.5.1 Integration of an aggregation attribute and several attributes

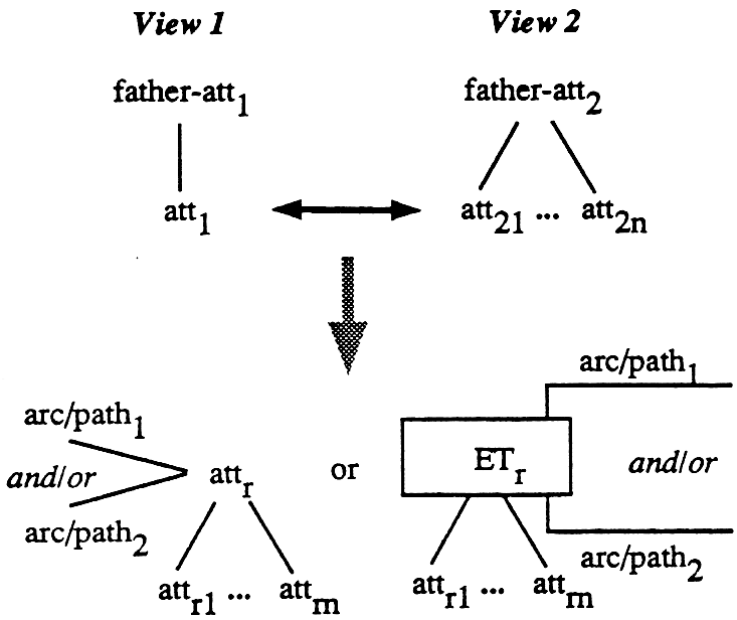


Figure 10.18 *- Attribute/Attributes aggregation integration*

This situation corresponds to the equivalence between a class-attribute att1 and the aggregation of several class-attributes att_{21}, ..., att_{2n}. It will be used very frequently for the integration of physical-oriented data structures.

This equivalence requires that the length and format of att1 will be equal to the length and format of the aggregation of att_{21}, ..., att_{2n}.

The result will be an attribute or an entity type (if it is forced by the arcs integration of att1×father-att1 and (att_{21}, ..., att_{2n})×father-att_2), with subattributes equal to att_{21}, ..., att_{2n}.

10.7.5.2 Integration of a grouping attribute and several attributes

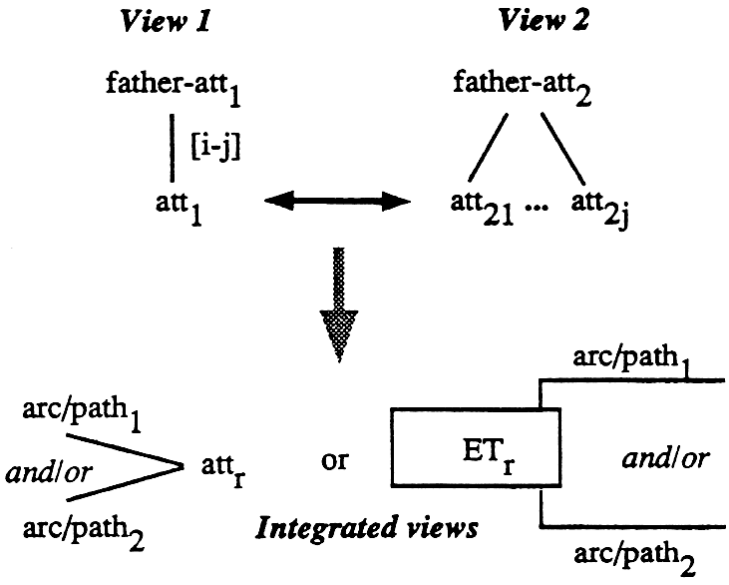


Figure 10.19 *- Attribute/Attributes grouping integration*

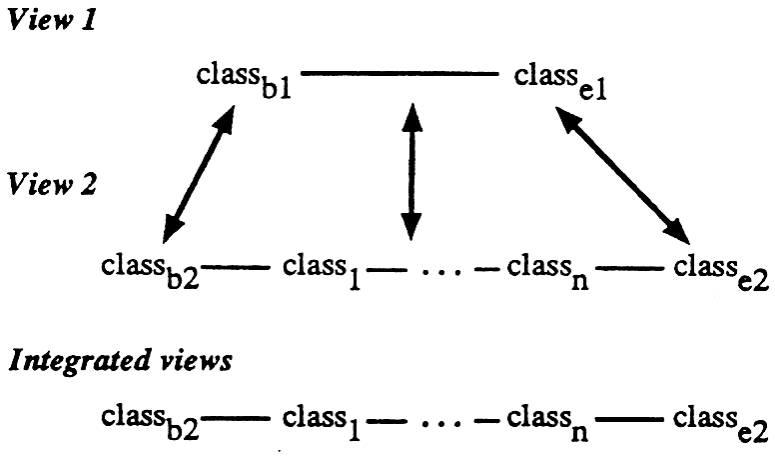
When a grouping attribute att is equivalent to several attributes att_{21}, ..., att_{2j}, that means the length and format of att_1 must be equal to those of the concatenation of att_{21}, ..., att_{2j} (usually j times the length and format of an att_{2i}).

The result can be :

- an attribute (or an entity type, according to the path integrations of att_1×father-att_1 and (att_2, ..., att_{2j})×father-att_2) equal to att_1. This result will have priority because the view 2 is often a less explicit one.
- several attributes equal to att21, ..., att2j, if this view is preferred.

10.7.6 Path/Path integration

10.7.6.1 Arc/Multi-arc path integration



In the case of an arc/path equivalence, they relates the same classes but the path contains more detailed structures (intermediary classes and arcs), which are not present in the arc. These detailed structures should be present in the result, so the path will be chosen as the resulting structure.

10.7.6.2 Non-arc path/Non-arc path integration

This complicated case occurs when two non-arc paths are equivalent but there is no more elementary correspondences between their components.

The only way to manage such a case is to have both paths in the integrated result, applying the techniques used for "no-correspondence" cases.

10.7.7 Integration of a grouping path and several paths

A path can be put in correspondence with several paths if it is a grouping of these paths (see figure 10.7 for an example).

Because the two structures are equivalent, the result of the integration can be either the grouping path, or only the different paths. However, we have also to use the rule for arc/multi-arc path integration. So, for our example, we will choose in priority the second view.

10.7.8 Restructuring step

As for the pre-integration work, because of the variety of the situations to manage, it is possible that, in some cases, the resulting integrated structure will not be exactly the expected result. The restructuration work will consist in this case in post-correcting the result, usually in applying some of the transformations defined in chapter 7. This process can also be considered as a component of the conceptualization work.

10.7.9 Processing our example

According to the correspondences asserted in 10.6.6 and using the principles we have explained, the integration of views 1 and 2 is made as following :

- the integration of the entity types ORDER1 and ORDER1 gives an entity type, whose name is ORDER1, for instance.
- the integration of the attributes ord-num in view 1 and ord-num in view 2 gives the attribute ord-num. The identifying characteristic of ord-num is conserved.
- the result of the integration of the repeating attribute ord-dates in view 1 and ord-date-cus, ord-date-fur in view 2 is the repeating attribute ord-dates.
- the attribute cus-num in view 1 corresponds to an attribute without name in view 2. Their integration produces an attribute which has the name cus-num.
- finally, the integration of cus-adr in view 2 with the four attributes cus-street, cus-nb, cus-zip, cus-city gives the compound attribute cus-adr made up with these four subattributes.

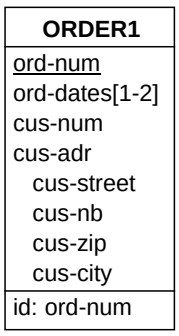


Figure 10.20 *- Result of the integration of views 1 and 2*

The integration of this result (let us call it view 4) with view 3 involves more arc/path integration (the correspondences defined in 10.6.6 between views 1 and 3 are almost the same) :

- the integration of the entity types ORDER1 and ORDER gives an entity type ORDER. The integration of their corresponding attributes ord-num and number gives the attribute number.

- the entity type CUSTOMER in view 3 does not exist in view 4. It is added. The relationship type SENDING does not exist too. It is also added.

  Another way to consider the case of CUSTOMER and SENDING is the integration of the arcs ORDER1×cus-num (or cus-adr) with ORDER×SENDING×CUSTOMER× number. The more detailed path and its intermediary classes (i.e. CUSTOMER and SENDING) must be choosen.

- the attribute ord-dates in view 4 has to be integrated with date of SENDING and date-fur of ORDER. We choose here the specific representation, keeping these two attributes. The arc ORDER×ord-dates[1] in view 4 corresponds simply to the arc ORDER×date-fur in view 3, but the arc ORDER×ord-dates[2] corresponds to the path ORDER×SENDING×date.

- cus-num of ORDER1 is integrated with number in CUSTOMER. The arc ORDER×cus-num corresponds to the path ORDER×SENDING×CUSTOMER×number. So the result is an attribute number for CUSTOMER.

- cus-adr in view 4 has to be integrated with address in view 3. As already mentionned, the path ORDER1 × cus-adr corresponds to ORDER × SENDING × CUSTOMER × address.

  The attributes cus-street, cus-nb, cus-zip, cus-city in view 4 do not exist in view 3. They are simply added to the attribute address of CUSTOMER.

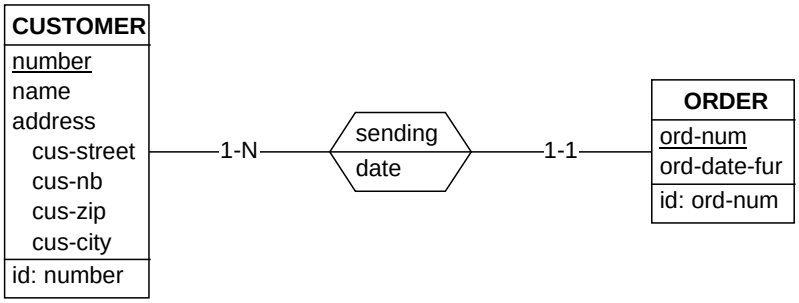


Figure 10.21 *- Result of the integration of views 1/2 and 3*

Chapter 11

DATA STRUCTURE EXTRACTION

This chapter describes the different processes that will be carried out in order to retrieve an abstraction of the DMS data structures used by a set of programs, i.e. the optimized, DMS-compliant schema.

11.1 INTRODUCTION

11.1.1 Objectives of the data structure extraction step

As analysed in chapter 5, the main objective of this phase is finding the logical, optimized DBMS-compliant schema of the database from the source programs.

Some additional objectives can be defined, such as finding the relationships between the programs and the database.

According to the framework developed in chapter 5, the data structure extraction appears as the first process in the reverse engineering activity (Figure 11.1).

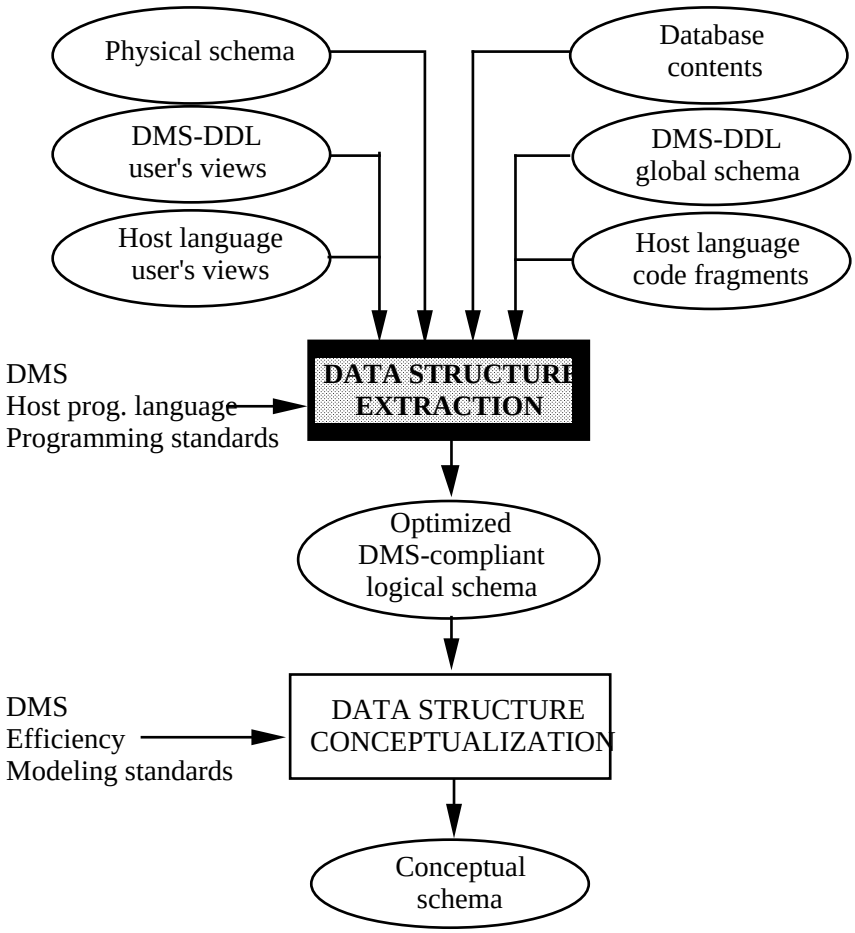


Figure 11.1 *- The data structure extraction phase as the first process of database reverse engineering.*

11.1.2 General strategies for data structure extraction

Before going into deeper details about the problems and related activities pertaining to this first process, we will layout a general structure for the various strategies that can be proposed for extracting the DMS schema of a database. Though they can be essential in some complex problems, some ancilliary activities will not be mentioned in this section. They will be developed in the next sections of this chapter.

`DATA STRUCTURE EXTRACTION` is strongly dependent on the DMS and on programming practices. An important criterion is the availability of a description of the global schema of the database. Figure 11.2 describes a realistic strategy dedicated to recovering the COBOL description of standard files. Figure 11.3 proposes a strategy intended to data structures for which a global description is available. In both cases, the process yields a complete description of the data structures. Most standard file managers do not offer a unique description of the file structures. Instead, these structures are scatered and fragmented throughout the application programs as DDL-DMS users' views. This observation dictates the following process organization.

- `VIEW EXTRACTION` is the front process that is to recover the complete views through which each application program perceives the data it is concerned with. The only available specification is the partial users' views coded as COBOL record description in the DATA DIVISION. This specification is partial, not only because it describes a part only of the file structure, but also due to *structure hiding*. Figure 11.4 illustrates how hidden structures can be elicited. In addition, the non-DMS parts of the schema, e.g. additional integrity constraints such referential integrity, must be recovered through the analysis of the procedural sections of the application programs, user interface, etc. `VIEW EXTRACTION` basically is the inverse of the `User's views CODING` forward process. In addition, it includes searching for the non-DMS parts of the logical schema.

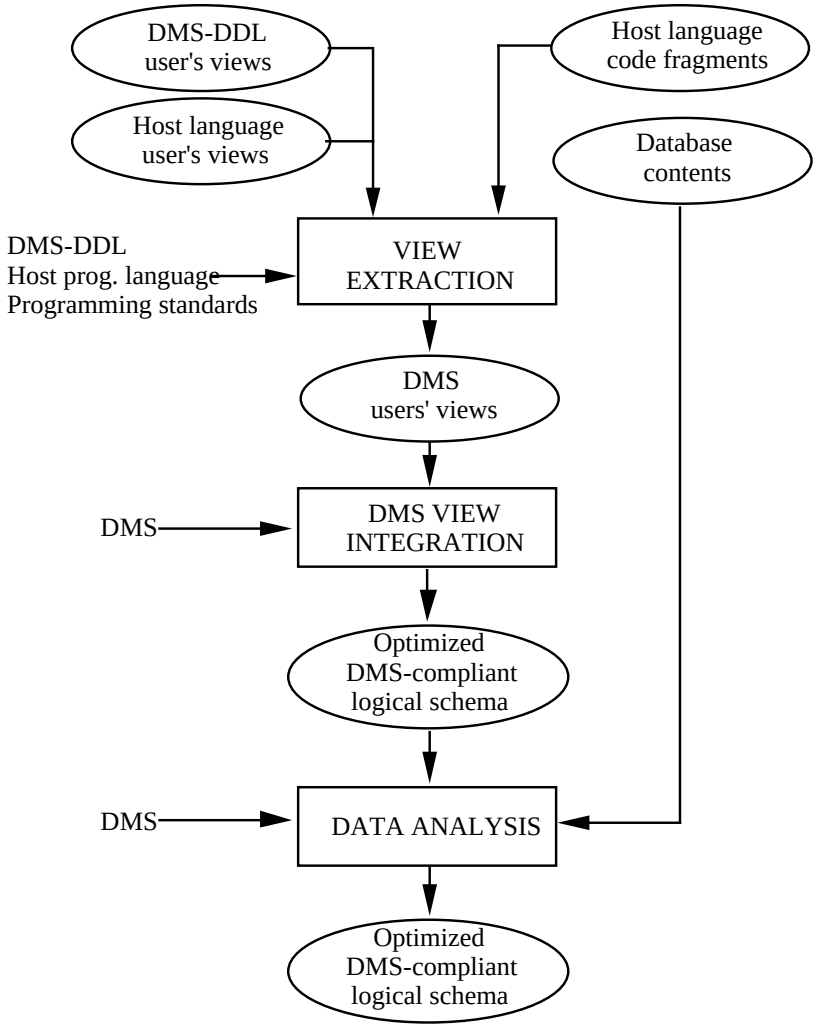


Figure 11.2 *- Strategy for data structure extraction process of standard file structures.*

- These views are then integrated by the `DMS VIEW INTEGRATION` process. Specific techniques are needed when compared with the traditional view integration problem (see chapter 10). Indeed, four important differences must be considered. First, the views are known to describe a unique and consistent collection of files. Second, the views are physical, and not only conceptual; therefore the integration rules must account

for DMS-specific and performance-oriented constructs. Third, the physical layout of the data gives essential hints to conduct the integration process (e.g. starting position of alternate fields), leading to specific integration rules. Finally, a view generally contains much redundancy, since files can be given both input and output structures. This process can be considered as the inverse of `User's views DERIVATION`. This process is discussed in appendix 1.

- The resulting global schema can then be further refined through `DATA ANALYSIS`, during which the file contents is examined to discover hints for, or to confirm, identifiers, foreign keys or field substructures for instance. This process has no forward counterpart.

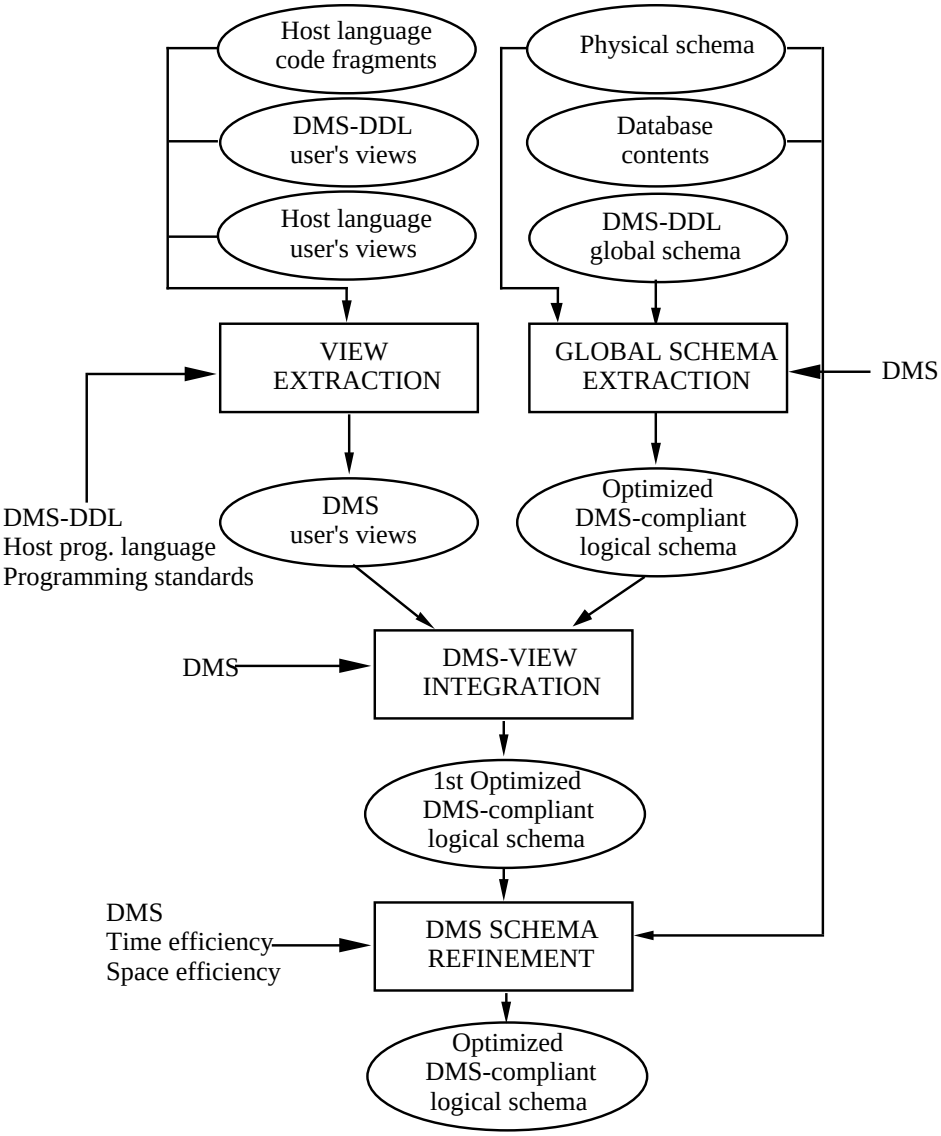


Figure 11.3 *- Strategy for data structure extraction process of DBMS structures.*

On the contrary, most DBMS (and some standard file managers as well, through some kind of data dictionary) make the global schema of the DB available, either as a DDL

program (IMS, CODAYL, SQL) or as data dictionary contents (SQL). Though this schema is not comprehensive, it makes a solid basis for further refinement. The approach is then different, since the users views analysis is now a secondary process.

- GLOBAL SCHEMA EXTRACTION is a rather straighforward process that parses the source DDL program (or any computerized form of it) in order to detect the specified data structures. It appears as the inverse of a part of DMS-DDL/Host CODING.
- For some DMS with unsophisticated view features (e.g. CODASYL), VIEW EXTRACTION is similar to what has been proposed for standard files. For others (e.g. SQL), it can give more important hints on hidden constraints. For instance, a relational view based on a join may suggest an undeclared foreign key. This process is the inverse of User's view CODING and of a part of DMS-DDL/Host CODING.
- DMS-VIEW INTEGRATION uses the views descriptions to refine the global schema obtained so far. This integration is physical as discussed hereabove.
- Through DMS SCHEMA REFINEMENT the physical schema and the database contents are examined in order to discover hints that may help refine the schema. For instance, a cluster defined on a join, or a non-unique index may suggest the existence of a foreign key.

```
program file
     01 CUSTOMER
         02 C-KEY  pic X(14)
         02 filler pic X(57)

working storage section
     01 IN-CUST
        02 ZIP-CODE pic X(8)
        02 SER-NUM  pic 9(6)
        02 C-DATA   pic X(57)

     01 EXPLODE1
        02 NAME     pic X(15)
        02 ADDRESS  pic X(30)
        02 ACCOUNT  pic 9(12)

procedure division
      ...
   read CUSTOMER into IN-CUST.
      ...
   move C-DATA of IN-CUST into EXPLODE1.
      ...
```

Figure 11.4 *- Hidden structure elicitation. The original record decomposition is recovered through data flow analysis.*

11.1.3 Organization of this chapter

This chapter is not intended to further develop strategic aspects of the data structure extraction phase. Therefore, it has been considered that a goal-oriented organization will be more adequate than an process-oriented structure. In other words, this chapter will no longer suggest a sequence of operations that must be followed, but it will develop ways to tackle problems that have to be solved, whatever the order in which they will be taken into account in an actual process.

Four classes of problems have been identified.

- Sources preparation :
  - Finding the relevant sources of information
  - Ordering sources of information in analysis packages.
- Extraction and construction of DBMS-compliant and optimized sub schemas
  - Finding the external organization of the program (main-prg, subprg, files, ...)
  - Finding the entity types
  - Finding the attributes and their format
  - Finding the relationship types
  - Finding the access keys
  - Finding the identifiers
  - Finding other I.C
- Merging sub-schemas into a unique DBMS-compliant and optimized schema
  - Redundancy elimination
- Miscellaneous problems
  - Completing descriptions of programs
  - Keeping trace on the mapping between the different versions of the data structures.

11.2 FINDING THE RELEVANT SOURCES OF INFORMATION

The problem is to find all the possible information sources, and to sort them out in order to keep those which may include useful information.

External (name, location, responsible, documentation) and internal (through examination of some kind of summary of the program) information are needed in order to choose the relevant sources.

11.3 ORDERING THE RELEVANT SOURCES OF INFORMATION

Not all sources are equal as far as their information contents are concerned. The order in which they are examined can make the reverse engineering process easier or more difficult. Criteria must be used to select the next source to be processed. The position of the source within the application, its function, the number of files it make use of, are such criteria.

11.4 FINDING THE EXTERNAL ORGANIZATION OF THE PROGRAMS

The external links between the programs (such as the calling relations) and the data flows between programs are a by-product that can be easily defined, and that can help the reverse engineer in analysing the sources. In addition, it is an important end-product for some applications of reverse engineering such as Data Administration.

11.5 FINDING THE ENTITY TYPES

Roughly speaking, we can make the hypothesis that external data agregates such as record types, segment types, relational tables, etc, can be perceived as the DMS-DDL translation of a logical entity type[1].

However, finding the entity types can be complex when a unique DMS data structure has been overloaded by several descriptions, such as by REDEFINES or RENAMES statements in COBOL, or when records of a single type are stored into different internal data structures, suggesting that this DMS structure encompasses represents more than one

---

[1] As already quoted, we are concerned with technical entity types and not with conceptual entity types. Let us remind that we are dealing with concepts related with the optimized DMS-compliant schema. Some technical entity types will be recognized of a conceptual nature while others will disapear (i.e. they will be transformed) during the conceptualization phase.

logical entity type. In these cases, the detection of the entity types is closely related with the elicitation of the field structure.

11.6 FINDING THE FIELDS AND THEIR FORMAT

The field decomposition of the technical data aggregates found in 11.5 is an important, and sometimes difficult step in this phase. Two situations can be distinguished.

When structure hidding has been avoided, direct examination of the DDL text leads to the exact and complete field description.

On the contrary, if structure hiding has been practiced, a more in-depth analysis of the non-DMS parts of the application programs is necessary. Examining the external and internal control flows, the internal and external data flows, the entry forms and report structure, must lead to a finer definition of the field structure.

The problem is that structure hiding can be used with any DMS, including the most recent ones. In addition it is difficult to know in advance whether this principle has been applied.

11.7 FINDING THE RELATIONSHIP TYPES

Logical relationship types will be looked for in schemas of DMS that allow a direct representation of them, namely the so-called hierarchical and network DBMS[2]. In most cases, the relationship types will be binary and one-to-many.

11.8 FINDING THE ACCESS KEYS

Access keys are abstractions of indices, hash files, calc keys, etc. As such, the access keys are purely technical constructs that bear no semantics. However, a description of the database cannot be considered complete without their specification. In addition, as discussed in section 2.4, an access key is often related with integrity constraints such as the following.

- If an access key is declared with no duplicates (or equivalent), it is an identifier as well.
- An access key may be associated with reference fields, i.e. with fields the values of which designate records (*foreign keys* for instance). Indeed, reference fields most often materialize many-to-one relationship types. When a relationship type supports path types, the corresponding reference fields are generally declared as an access key.

---

[2] More recent DBMS also fall in this category. Let's mention Object-Oriented DBMS for instance. OO schemas will probably not be reverse engineered before several years.

11.9 FINDING THE IDENTIFIERS

Their detection is easy when they are structurally declared in the DMS schema.

Not all kinds of identifiers can be managed by all DMS. Therefore, detecting them can sometimes be considered as hidden structure elicitation. Let's consider some examples.

- Sequential file managers cannot take this notion in charge since they generally require an associated access key.
- CODASYL DBMS allow two kind of identifiers, namely (1) identifying a record among the records of a given type in a given file (only one per record type), (2) identifying the records among the members of a set of a given type (only one per set type). All other identifiers cannot be implemented explicitly.
- In IMS databases, any non-root segment can be identified among the children of its parent segment. An identifier that does not fall in this category cannot be represented explicitly.
- In TOTAL/IMAGE databases, detail records (variable entry data sets) cannot have an identifier.

As hidden structures, implicit identifiers will be retrieved either through hints based on their name for instance, or by examination of the procedural parts of the programs that process the concerned entity type. When available, the database contents can be a interesting source of information as well.

11.10 FINDING THE OTHER INTEGRITY CONSTRAINTS

Following the identifiers, the next most important integrity constraint is the **referential constraint**. It can be explicitly managed by the DMS, such as in recent relational DBMS, either through explicit declaration (DB2, SQL/DS), or through triggers (SYBASE), or through checking predicates (VAX RDB), or through updatable views (ORACLE).

In other cases, this constraint must be considered as a hidden structure. Its recovering will be based on hints such as the name structure of the reference fields, modification patterns in the procedural parts of the application programs, event-driven procedures in entry forms (ORACLE), database contenrs examination, etc.

Note that this constraint can be found in schemas of hierarchical and network DBMS as well.

Other important constraints are related with the existence, within a schema, of **structural redundancies**, of **non-key functional dependencies** (leading to unnormalized structures), **exclusive attributes/roles**, and **value dependencies**. Except in some relational DBMS (VAX RDB, SYBASE), there is no explicit means to declare such constraints. Moreover, even with such DBMS, they can remain undeclared for simplicity and performance reasons.

Therefore, they can be considered, in most cases, as hidden structures. They will be retrieved by name structures and by other ways as mentioned hereabove.

11.11 VIEW INTEGRATION / SCHEMA REDUNDANCY REDUCTION

Most DBMS provide an explicit representation of the global schema , either in the DDL format (CODASYL systems), or in some kind of dedicated data dictionary (SQL systems). Some lower level DMS can run in an environment that provide a centralized repository for data structure descriptions (VAX CDD). In these situations, the DDL texts or the contents of the data dictionary/repository allow retrieving the global DMS schema in a rather straightforward way.

The situation is quite different when a low-level DMS is used in a traditional way, i.e. when the only traces of the data structure description are to be found in the application programs themselves, in a form we have named coded user's views.

Working out the source text of a program leads to retrieving a DMS-compliant user's view. When several programs have been analysed, the designer is faced with a collection of user's views. According to the strategy of early integration, he will try to reduce the number of views in order to obtain the global DMS-compliant schema of the database. This strategy is particularly useful when the collection of views is both fairly large and highly redundant.

On the other hand, the late integration strategy propose to conceptualize each user's view, then to integrate them once they have a conceptual status.

Warning

DBMS can be used, at least partially, the same way as low-level DMS are used. Indeed, the DBMS schema can present data structures that are so grossly defined that they give little semantic information about the data they organize. This would be the case for IMS unstructured segment types that are redefined several times according to different decompositions. These situations could present some problems close to those which have been presented hereabove.

11.12 COMPLETING THE DESCRIPTION OF THE PROGRAMS

The gross description that was aimed at guiding the reverse engineer (step 1.4), can be augmented with refined information on , e.g., the entity-type / programs links obtained so far. For instance, this information can tell which program units create CUSTOMER entities, and which ones read INVOICE entities.

11.13 KEEPING TRACE OF THE MAPPING

Though few transformations have been carried out in the activities described above, the relation between the components of the DMS schema and their physical sources can be far from simple.

11.14 DATA STRUCTURE EXTRACTION OF RELATIONAL DATABASES

We will give the main translation rules for expressing the RELATIONAL-compliant optimized schema from the source text fragments that declare the record structures.

11.14.1 RELATIONAL translation rules

RELATIONAL declaration	ER expression
• `create schema (or database)`	schema
• `create dbspace`	collection (of entity types)
• `create table`	entity type
`within`	in a collection
• *`column`*	single-valued, atomic attribute of the entity type
`not null`	mandatory (otherwise optional)
• `create index`	access key
• `primary key`	identifier (+ access key in some DBMS)
• `foreign key`	inclusion (referential) constraint
• `unique`	identifier
• `create unique index`	access key + identifier
• `check`	integrity constraint (requires further analysis)
• `trigger`	requires further analysis : generally integrity constraints

11.14.2 RELATIONAL example

The following example illustrates these translation rules. A more complete example is studied in chapter 15.

SQL schema

```
create table CUSTOMER (
                        CNUM char(6) not null primary key,
                        CNAME char(25) not null,
                        AD_NUM decimal(6) not null,
                        AD_STREET char(35) not null,
                        AD_CITY char(40) not null)

create table ORDER (
                        CNUM char(6) not null,
                        ONUM char(8) not null,
                        DATE date not null,
                        primary key (CNUM,ONUM),
                        foreign key (CNUM) references CUSTOMER )

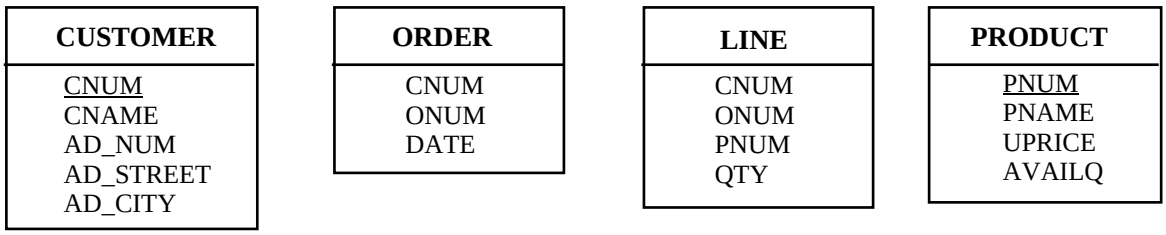
create table LINE (
                        CNUM char(6) not null,
                        ONUM char(8) not null,
                        PNUM char(15) not null,
                        QTY decimal(6) not null,
                        foreign key (CNUM,ONUM) references ORDER,
                        foreign key (PNUM) references PRODUCT )

create table PRODUCT (
                        PNUM char(15) not null primary key,
                        PNAME char(25),
                        UPRICE decimal(5,2),
                        AVAILQ decimal(6) )

create index XCNAME on CUSTOMER(CNAME)
```

Relational-compliant optimized schema

The ER expression of this schema is given in Figure 11.5. We have supposed that a unique index is automatically defined on each primary key.



id(ORDER) : CNUM,ONUM

ORDER.CNUM is-in CUSTOMER.CNUM
LINE.(CNUM,ONUM) is-in ORDER.(CNUM,ONUM)
LINE.PNUM is-in PRODUCT.PNUM

key(CUSTOMER) : CNUM
key(CUSTOMER) : CNAME
key(ORDER) : CNUM,ONUM
key(PRODUCT) : PNUM

Figure 11.5 *- ER expression of the relational-compliant schema of the database.*

11.15 DATA STRUCTURE EXTRACTION OF COBOL FILES

We will give the main translation rules for expressing the COBOL-compliant optimized schema from the source text fragments that declare the record structures. The extraction of additional properties are illustrated in the case study, chapter 16.

11.15.1 COBOL translation rules

COBOL declaration	ER expression
• *set of files*	schema
• `select`	collection (of entity types)
• `records are`	entity types
	their inclusion into the collection
• `organization sequential`	-
`indexed`	-
`relative`	technical attribute made identifier + access key
• `record key`	identifier + access key
• `alternate record key`	access key + identifier

`with duplicates`	objection : the key is not an identifier !
• `01 record-name`	the description of the entity type follows
• `02 field-name`	attribute of entity type
`... picture (or usage)`	atomic field
`... occurs`	multivalued field
• `03 field-name`	component of a compound attribute

11.15.2 COBOL example

The following short example illustrates these translation rules. A more complete example is studied in chapter 15.

COBOL schema

```
select CUSTOMERS assign to DSK:FC ;
organization is indexed;
record key is CNUM
alternate record key is CNAME with duplicates.

select ORDERS assign to DSK:FO ;
organization is indexed;
record key is OID.

select PRODUCTS assign to DSK:FP ;
organization is indexed;
record key is PNUM.


DATA DIVISION.
FILE SECTION.

fd CUSTOMERS;  label record is standard.
   record is CUSTOMER.
   01  CUSTOMER.
       03 CNUM pic is X(6).
       03 CNAME pic is X(25).
       03 ADDRESS.
          05 NUM pic is 9(6).
          05 STREET pic is X(35).
          05 CITY pic is 9(40).

fd ORDERS;  label record is standard.
   record is standard.
   01  ORDER.
       03 OID.
          05 CNUM pic is X(6).
          05 ONUM pic is X(8).
       03 DATE pic is X(6);
```

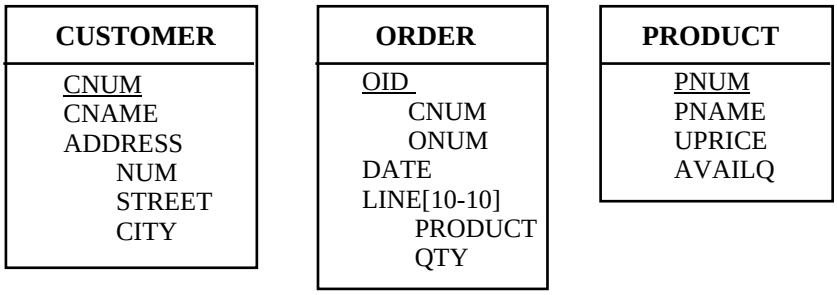
```
     03 LINE occurs 10.
           05 PRODUCT pic is X(15).
           05 QTY pic is 9(6).

fd PRODUCTS;  label record is standard.
   record is PRODUCT.
   01  PRODUCT.
       03 PNUM pic is X(10).
       03 PNAME pic is X(25).
       03 UPRICE pic is 9(5)V9(2).
       03 AVAILQ pic is 9(6).
```

COBOL-compliant optimized schema

The ER expression of this schema is given in Figure 11.6. Further source program analysis would surely have made other properties explicit :

- ORDER.CNUM as a reference to CUSTOMER;
- ORDER.PRODUCT as a reference to PRODUCT;
- cardinality of ORDER.LINE [1-10] instead of [10-10].



key(CUSTOMER) : CNUM
key(CUSTOMER) : CNAME
key(ORDER) : OID
key(PRODUCT) : PNUM

Figure 11.6 *- ER expression of the COBOL-compliant schema of the database.*

11.16 DATA STRUCTURE EXTRACTION OF CODASYL DATABASES

Here follows the main translation rules for expressing the CODASYL-compliant optimized schema from the source text fragments that declare the record structures.

11.16.1 CODASYL translation rules

CODASYL declaration	ER expression
• `schema name is ..`	schema
• `area name is ..`	collection (of entity types)
• `record name is ..`	entity types
`within`	in a collection
`location calc`	access key
`duplicates not`	+ identifier
• *`items definition`*	*see COBOL*
• `set name is ..`	binary rel-type
`owner is ..`	the [0-N] role
`member is ..`	the [0-1] or [1-1] role
`- mandatory automatic`	- [1-1]
`- optional manual`	- [0-1]
`duplicates not for`	hybrid identifier (role + attributes)

11.16.2 CODASYL example

The following short example illustrates these translation rules. A more complete example is studied in chapter 17.

CODASYL schema

```
schema name is ORDERS.

area name is AORDER.

record name is CUSTOMER
        location calc using CNUM, duplicates not allowed.
        2 CNUM pic is X(6).
        2 CNAME pic is X(25).
        2 ADDRESS.
          3 NUM pic is 9(6).
          3 STREET pic is X(35).
          3 CITY pic is 9(40).

record name is PRODUCT
        location calc using PNUM, duplicates not allowed.
        2 PNUM pic is X(10).
        2 PNAME pic is X(25).
        2 UPRICE pic is 9(5)V9(2).
        2 AVAILQ pic is 9(6).

record name is ORDER
        2  ONUM pic X(8).
        2  DATE pic X(6).

record name is LINE.
        2 QTY pic is 9(6).

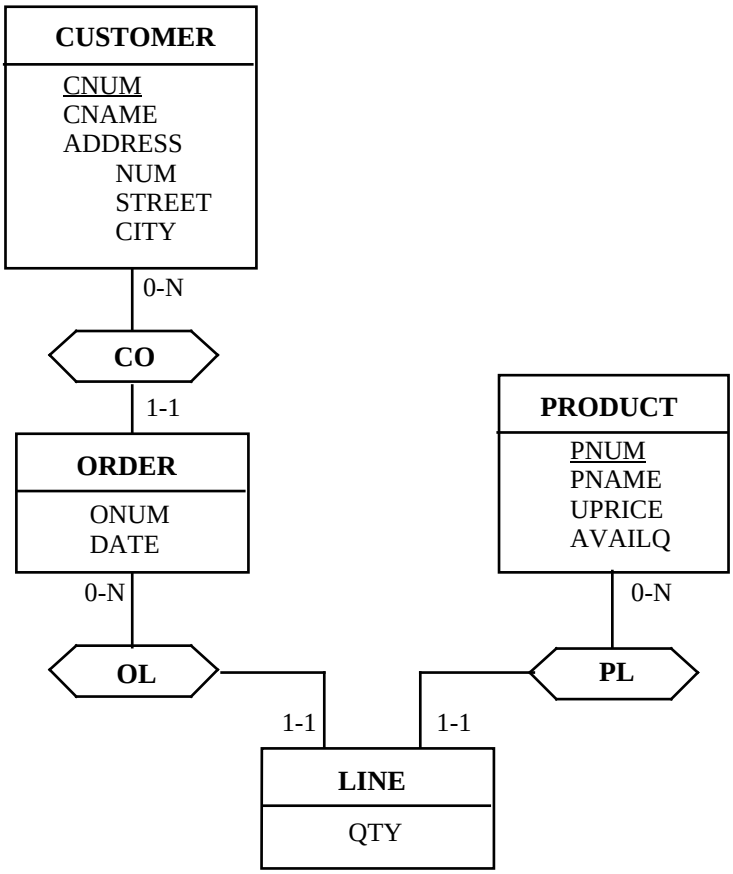
set name PL
        owner PRODUCT
        member LINE mandatory automatic.

set name OL
        owner ORDER
        member LINE mandatory automatic

set name CO
        owner CUSTOMER
        member ORDER mandatory automatic
        duplicates not allowed for ONUM.
```

CODASYL-compliant optimized schema

The ER expression of this schema is given in Figure 11.7.



id(ORDER) : CUSTOMER,ONUM

Figure 11.7 *- ER expression of the CODASYL-compliant schema of the database.*

Chapter 12

DATA STRUCTURE CONCEPTUALIZATION

This chapter describes the processes that can be carried out in order to extract a possible conceptual schema from a DMS schema.

12.1 INTRODUCTION

12.1.1 Objective of the data structure conceptualization phase

Data structure conceptualization appears as the second process in the general database reverse engineering strategy proposed in chapter 5 (Figure 12.1). Its goal is the production of a normalized conceptual schema that is the natural expression of the semantics of the data structures elicited during the Data Structure Extraction process.

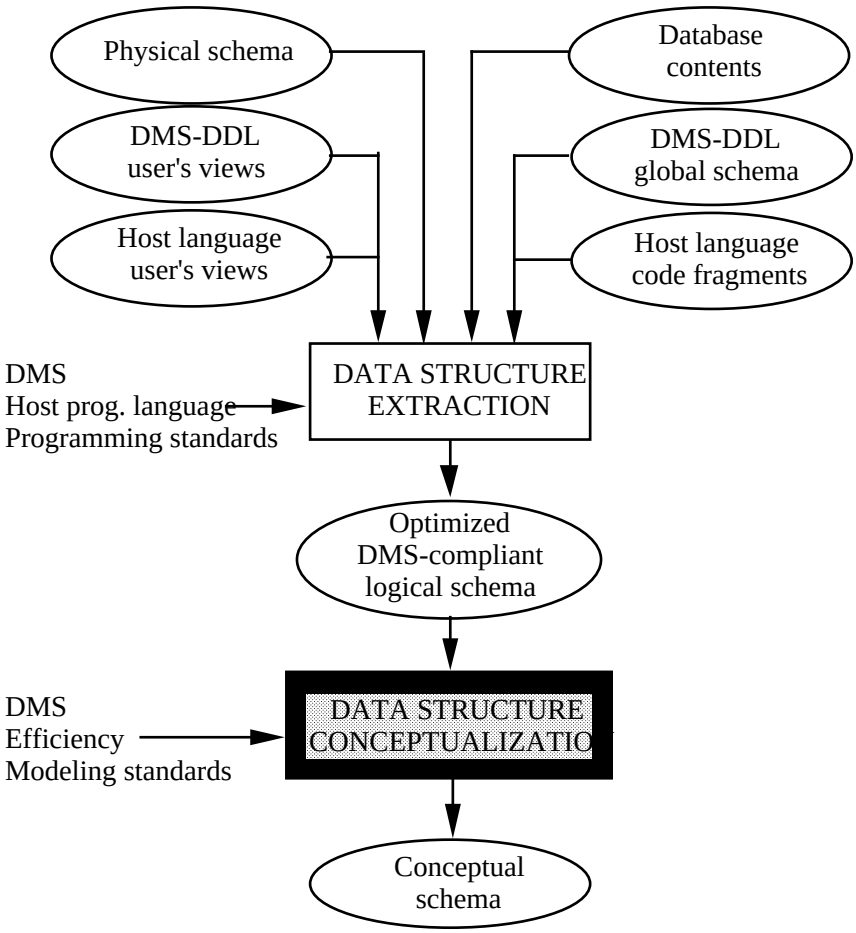


Figure 12.1 *- The data structure conceptualization phase is the second process of database reverse engineering.*

DATA STRUCTURE CONCEPTUALIZATION, can be decomposed into two major sub-processes, namely BASIC CONCEPTUALIZATION and CONCEPTUAL NORMALIZATION (figure 12.2).

- BASIC CONCEPTUALIZATION is a highly DMS-dependent process that is intended to clean the schema from the DMS-specific and performance-oriented constructs. It produces the Basic Conceptual schema, that is an unsophisticated conceptual description of the database. For instance, this schema is mainly made of one-to-many rel-types, and includes no IS-A hierarchy. Its aim is to reverse the effect of DMS-dependent and DMS-independent optimization, and of DMS translation processes.
- CONCEPTUAL NORMALIZATION is an optional, DMS-independent, process through which the schema is further refined and structurally improved in order to improve its readability, and to make it comply with the corporate methodological standards. In particular, it includes reversing the effect of the SCHEMA SIMPLIFICATION process. It has the same objectives (minimality, clarity, normalization, etc) and uses

the same techniques as the CONCEPTUAL NORMALIZATION process that has been described in section 3.8 as a sub-process of CONCEPTUAL DESIGN.

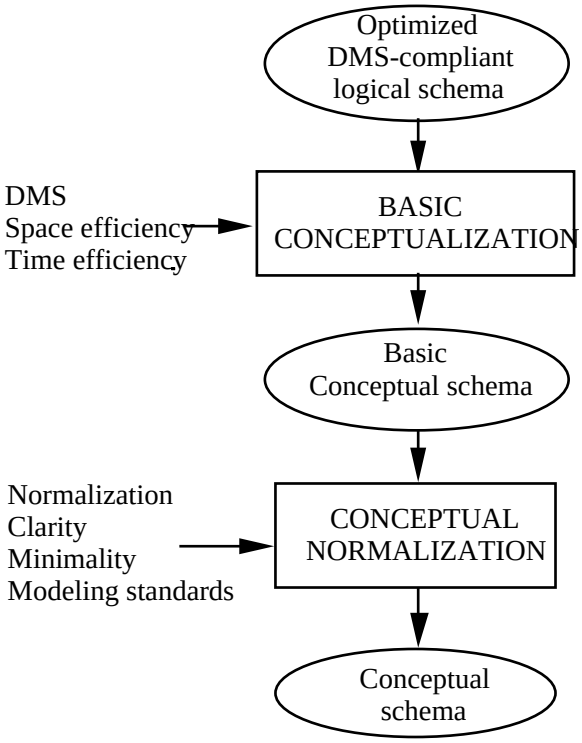


Figure 12.2 *- Strategy for data structure conceptualization.*

BASIC CONCEPTUALIZATION is a critical process that requires advanced knowledge and experience in database technology and design, together with strong domain knowledge. According to the analysis proposed in section 3.8 and in chapter 5, it can be decomposed into three processes (Figure 12.3). Though they are generally conducted in parallel, these processes are considered as distinct since they may use different techniques, knowledge and reasonings.

- DMS-UNTRANSLATION is the inverse of DMS-TRANSLATION forward process. It consists in interpreting DMS-dependent constructs as the implementation of a conceptual construct to discover.
- DMS-SPECIFIC DE-OPTIMIZATION is the inverse of the DMS-dependent OPTIMIZATION forward process. It is based on the knowledge of the physical characteristics of the DMS and on the requirements that have been considered when building the database.
- DMS-INDEPENDENT DE-OPTIMIZATION is the inverse of DMS-independent OPTIMIZATION forward process. It is based on generic knowledge on database performance and on the requirements that have been considered when building the database.

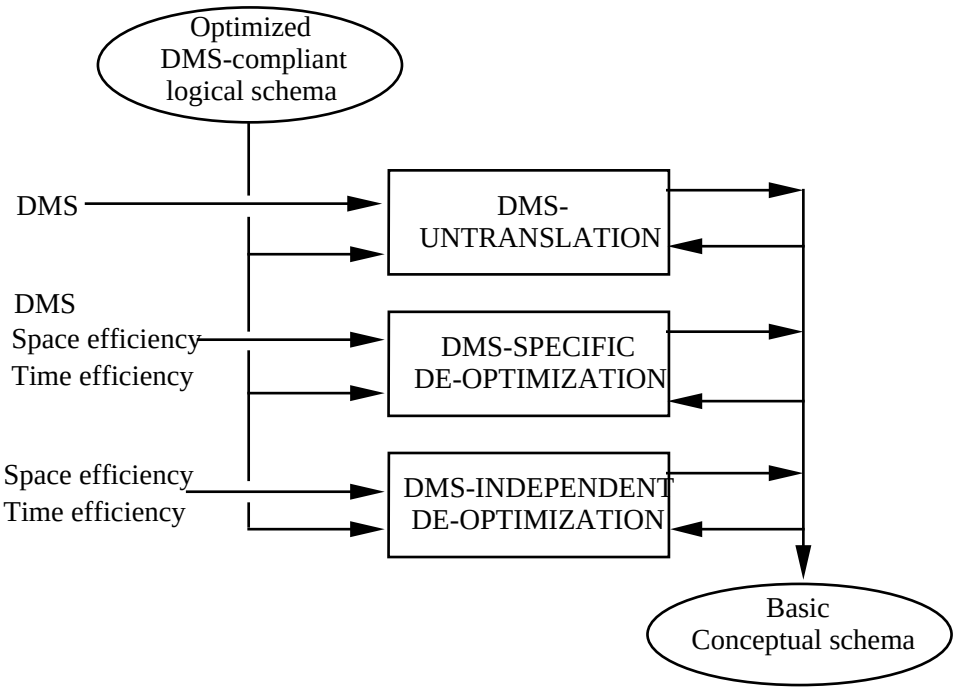


Figure 12.3 *- Strategy for basic conceptualization.*

`CONCEPTUAL NORMALIZATION` consists in restructuring the basic conceptual schema in order to give it additional characteristics pertaining to criteria such as clarity, minimality, readability or compliance with in-house methodological or development standards. Indeed, the Basic Conceptualization process produces a conceptual schema that may look somewhat awkward according to the current *methodological culture* of the company. For instance, concepts that should be represented by N-ary and many-to-many relationship types are represented by entity types and one-to-many relationship types.

The following list mentions the main constructs that can be restructured in order to make the basic conceptual schema more readable :

- A **relationship entity type**, i.e. an entity type whose aim obviously is to relate two or more entity types, can be transformed into a rel-type.
- An **attribute entity type**, i.e. an entity type that has one attribute only, and is linked to one entity type E only, can be replaced by a simple attribute of E.
- A **one-to-one rel-type** may express the connection between fragments A1 and A2 of a unique entity type A (vertical splitting). These fragments can be merged into this entity type.
- An entity type that appears as comprising **too many attributes** and roles can suggest a decomposition into fragments linked by *one-to-one* rel-types.
- A **N-ary rel-type that has a role with cardinality 1-1** can be decomposed into binary, one-to-many rel-types.
- A collection of entity types that seem to have some **attributes and roles in common** can be made the subtypes of a common supertype that inherits the common characteristics.

- An entity type that has **exclusive, optional, subsets of attributes and roles** can be given subtypes, each of them inheriting one of these groups. An entity type that has a group of optional attributes and roles can also be examined for such a transformation.
- One or several **one-to-one rel-types** that concern a common entity type A may also express a specialization relation in which A is the supertype. These rel-types are replaced by IS-A relations.

Additional activities can be proposed in order to complete the conceptual schema (see section 5.4.3). Though they are not specific to reverse engineering only, they can be particularly important in this context, due to the complexity and incompleteness of the sources of information.

- **Name processing**. The names of the objects can be revised, for instance in order to make them comply with the naming conventions of the organization.
- **Semantic interpretation**. Each of the objects of the schema must be given a clear and complete description that defines its meaning in the context in which it will be used.
- **Explicitation of hidden semantic constructs**. In the reverse engineering of an application, some information cannot be deduced nor retrieved from the source texts alone. Such informations could be found in other sources of reverse engineering and added to the conceptual schema of an application. One of the main sources is the domain knowledge. These hidden semantic constructs are to be distinguished from the hidden structures that have been discussed in chapter 11. Indeed, hidden structures are schema objects, including integrity constraints, that cannot be found in the DMS expression of the schema, but such that there exist some traces of them in the source texts of the program for instance. A hidden structure is described in the source text, but in an obscure way. A hidden semantic construct is in no way expressed in the source texts that have been analyzed. The difference between them is sometimes loose, since a construct will fall in one category or the other depending on the skill of the analyst, or on the availability of the relevant documents.
- **Enrichment**. Enrichment consists in adding, suppressing or modifying elements in a conceptual schema in order to make it closer to the business model of the application. These additional elements are not represented in the current database. Therefore the resulting conceptual schema is no longer a strict description of the semantics of this database, but rather is a superset of this semantics.
- **Schema integration**. There are two ways to deal with a collection of user's views extracted from, say, each application programs. The first way consists in merging them (in the Data Structure Extraction process) to produce a unique, global, DMS schema. The other way consists in keeping them separate, then in conceptualizing each of them into a conceptual user's view, and finaly in merging the latters into a unique, global conceptual schema. Therefore, this second approach requires a final schema integration process. This process has been discussed in chapter 10.

The following sections of the chapter are organized as follows :

- section 12.2 discusses the problems of removing optimization constructs;
- section 12.3 develops reasonings and techniques for *un-translating* the data structures according to some of the most common DMS, namely COBOL structures, relational schemas, CODASYL schemas, TOTAL/IMAGE schemas and IMS schemas;
- section 12.4 presents the problem of conceptual normalization.

This presentation is mainly based on transformation techniques. Most of the transformation that will be used have been discussed in chapter 7, but, for simplicity, some others will be described in the technical annex.

12.2 *DE-OPTIMIZATION* OF A SCHEMA

Both DMS-dependent and DMS-independent optimization processes will be considered as a whole, since they make use of the same set of techniques, though possibly through different strategies. There are three major families of optimization techniques based on schema transformations, namely *unnormalization*, *structural redundancy* and *restructuration*. They must be precisely understood in order to reverse their effect on a schema. In particular, some of them are more specifically fitted for some DMS than for others.

Accordingly, the de-optimization techniques will be dedicated to reversing the effet of these practices. They will be presented, wherever possible, as schema transformation techniques.

Finally, let us mention additional techniques consisting in detecting and removing technical contructs that aim at improving the performance or the structure of the programs, e.g. counters, local variable states or level indicators in hierarchical structures.

12.2.1 Normalization

Optimization techniques

The most common unnormalization technique consists in merging two entity types linked by a one-to-many rel-type into a single entity type. This technique allows obtaining the information of both entity types in one logical access, thereby decreasing the access time. On the other hand, it induces redundancies.

Optimization detection

The principles of unnormalized structures and of their detection has been presented in section 9.9. We shall recall some elementary properties :

- An unnormalized structure is detected in entity type E by the fact that the determinant of a dependency F holding in the elements of E is not an identifier of E.

- In the same way, an unnormalized structure is detected in relationship type R by the fact that the determinant of a dependency F holding in the elements of R is not an identifier of R.

This detection requires the elicitation of the identifiers of object E or R, and of all the dependencies holding in E or R.

De-optimization techniques

Reversing this transformation consists in *normalizing* the entity type (the most common example) by segregating its components according to the example below.

Examples

The first example (figure 12.4) concerns an entity type. A functional dependency has been detected among the components of B. Its determinent is clearly not an identifier of B. Therefore, the components that appear in this dependency are extracted to form entity type B2, then connected to the initial fragment B1 by a one-to-many relationship type.

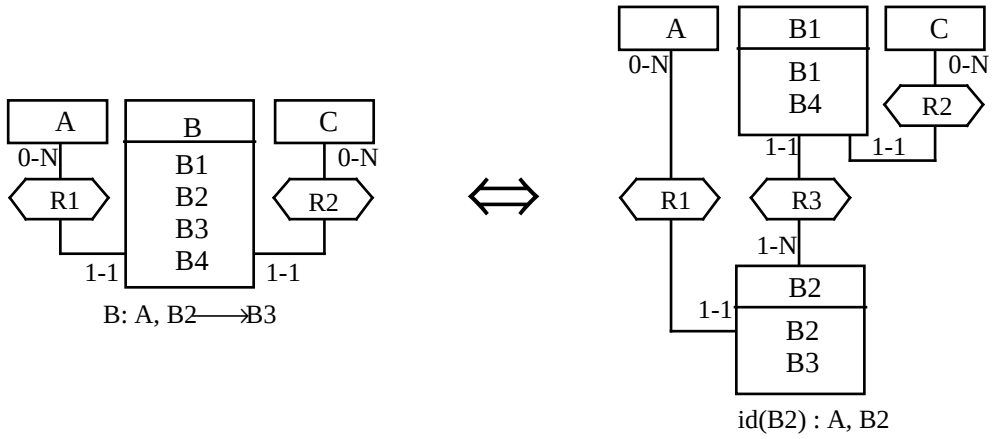


Figure 12.4 *- Normalization of an entity type*

The second example (Figure 12.5) is given for generality purpose. Indeed, this situation will appear in the Conceptual Normalization process rather than in Basic Conceptualization, where only binary relationship types appear. A functional dependency is detected among the attributes and roles of R. These components are partitioned according to the decomposition principles (section 9.9.2.5).

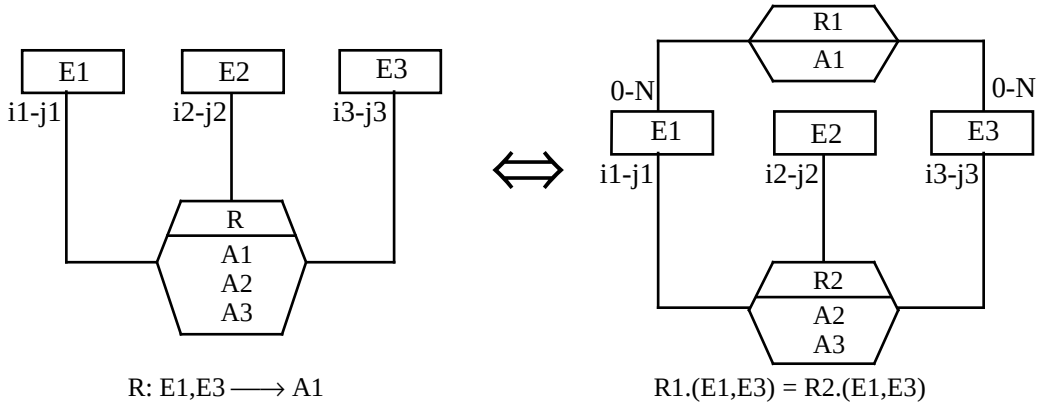


Figure 12.5 *- Normalization of a relationship type*

12.2.2 Structural redundancies removing

Optimization techniques

Structural redundancy techniques consist in adding new constructs in a schema such that their instances can be computed from instances of other constructs. Attribute duplication, rel-type composition and aggregate representation are some examples of common optimization structural redundancies. These transformations are (trivially) symmetrically reversible since they merely add derivable constructs without modifying the source constructs.

These techniques have been discussed in section 9.2 to 9.6, and this discussion will not be duplicated here. In particular,

- section 9.2 analyses the attribute/attribute redundancy,
- section 9.3 analyses the attribute/role redundancy,
- section 9.4 analyses the attribute/rel-type redundancy,
- section 9.5 analyses the entity type/rel-type redundancy,
- section 9.6 analyses the rel-type/rel-type redundancy.

Optimization detection

Section 9.2 to 9.6 propose heuristics to detect redundancy properties according to the five classes of techniques mentioned above.

De-optimization techniques

The technique consists in removing the redundant constructs. Let us recall that some situations may be more complex than suggested hereabove. In particular, the similarity between redundancy and integrity constraint (section 9.7) and partial redundancy (section 9.8) must induce a careful approach of this problem.

Examples

Figure 12.6 depicts the elimination of a composed rel-type and of a duplicate attribute.

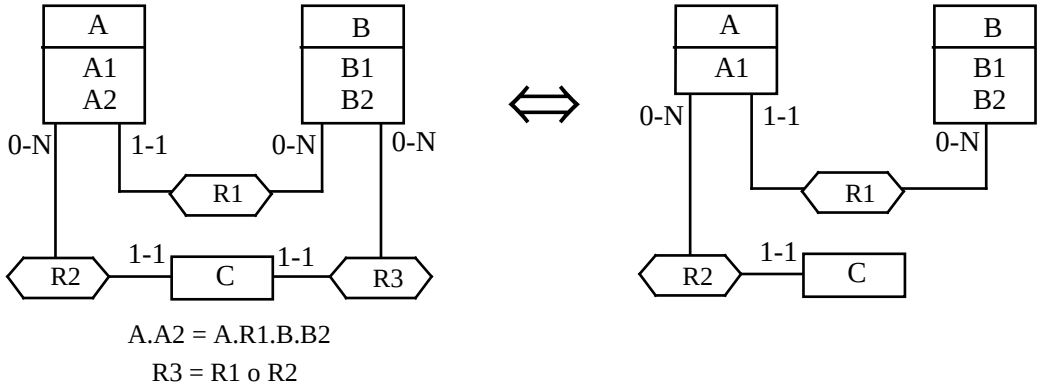


Figure 12.6 *- A2 has been recognized as a duplicate attribute and R3 as the composition of R1 and R2. They are removed.*

12.2.3 Restructuration

Restructuration consists in replacing a construct by other constructs in such a way that the resulting schema yields better performance. These techniques introduce no redundancy as did the other techniques (unnormalization and structural redundancy).

Almost all the semantics-preserving transformations that can be applied on simple schemas (i.e. without sophisticated constructs such as IS-A, N-ary relationship type, rel-types with attributes, etc) can be considered useful to gain better performances in some specific contexts. To keep things simple, we will mention six popular techniques that can be used to improve the performance quality of a database.

12.2.3.1 Representation of an attribute by an *attribute* entity type

Optimization technique

An attribute B1 of an entity type A is extracted, and expressed as an autonomous entity type B (the *attribute* entity type*)* whose unique, and identifying, attribute is B1. A one-to-many rel-type is defined between A and B. This technique is useful when B1 values are large and seldom used (see 7.7.1).

Optimization detection

The *attribute* entity type has one attribute, which in addition is its identifier. This entity type is mandatorily linked to one other entity type only through a one-to-many rel-type. The *attribute* entity type appears as nothing more than a property of the latter.

De-optimisation technique

Replace the *attribute* entity type by an attribute associated with the main entity type (see 7.2.3).

Example

Figure 12.7 illustrates how *attribute* entity type B is replaced by attribute B1 of A.

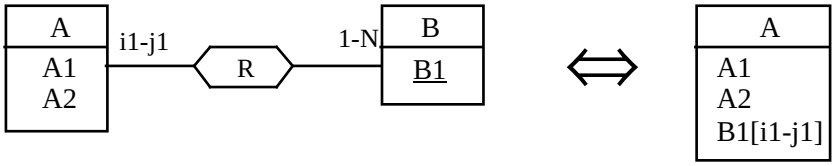


Figure 12.7 *- Replacement of an attribute entity type.*

12.2.3.2 Representation of an entity type by an attribute

Optimization technique

An entity type B linked to one other entity type A is represented by migrating its attribute(s) to A (see 7.2.3). Ideally, the identifier of B is made of all its attributes (otherwise the technique falls in the *unnormalization* category).

Optimization detection

A set of attributes of entity type A appears as the description of an external entity type.

De-optimisation technique

Transform these attributes into an entity type (see 7.7.1).

Example

Figure 12.8 shows how attribute B1 (possibly multivalued) is represented by a specific entity type.

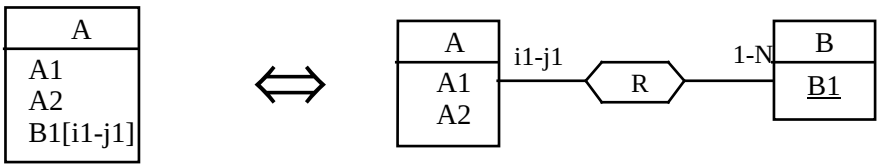


Figure 12.8 *- Transformation of an attribute into an entity type.*

12.2.3.3 Horizontal partitioning

Optimization technique

Horizontal partitioning consists in replacing entity type A by entity types AA and AB in such a way that the population of A is partitioned into those of AA and AB. This technique yield smaller data sets, smaller indexes and allows for better space allocation. However, it may generate complex integrity constraints such global identifiers holding not only on each resulting entity type, but more specifically on their union.

Horizontal partitioning can be applied to rel-types as well. For instance the population of *many-to-many* rel-type R can be split into *one-to-many* rel-type R1 that collects specific instances, while *many-to-many* rel-type R2 collects the other instances of R. The implementation of R1 will be more efficient than that of R2.

Optimization detection

Two entity types have similar names, the same structure (attributes, roles, constraints) and non overlapping identifier value sets.

De-optimisation technique

Define one entity type whose semantics covers those of the two origin entity types.

Example

In figure 12.9, entity types AA and AB are replaced by entity type A representing the union of their populations.

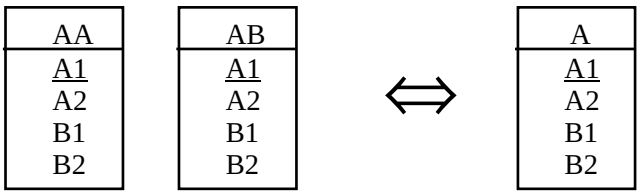


Figure 12.9 *- Merging two similar entity types.*

12.2.3.4 Horizontal merging

Optimization technique

The inverse of horizontal splitting is **horizontal merging**. It allows decreasing the number of entity types and making physical clustering easier.

Optimization detection

An entity type appears as the representation of the union of two (or more) real-world object sets.

De-optimisation technique

Replace the entity type with two similar entity types.

Example

Figure 12.10 shows how the population of entity type A is partitioned into those of AA and AB.

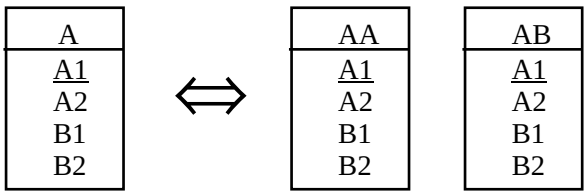


Figure 12.10 *- Splitting of an entity type.*

12.2.3.5 Vertical partitioning

Optimization technique

Vertical partitioning of entity type A partitions its attribute/role components into two (or more) entity types A and B, linked by a one-to-one rel-type R. This partitioning is generally driven by processing consideration : components that are used simultaneously are grouped into a specific entity type. This decreases the physical length of A entities, and improves access time (see 7.2.2).

Optimization detection

Two entity types linked by a one-to-one rel-type appear as complementary.

De-optimisation technique

Define one entity type only. This entity type inherits all the attributes, roles and constraints of the two source entity types (see 7.3.2).

Example

Figure 12.11 describes the merging of entity types A and B into the single entity type A. Two cases are considered : R is mandatory for A, and R is optional for A.

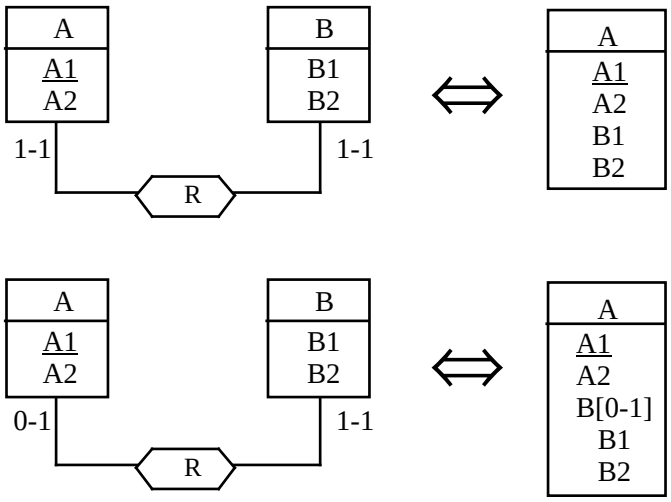


Figure 12.11 *- Merging of two entity types into a single one (two variants).*

12.2.3.6 Vertical merging

Optimization technique

Vertical merging is the inverse technique. It merges two distinct entity types that are linked by a one-to-one rel-type in order to avoid double access to get related entities (see 7.3.2).

Optimization detection

An entity type includes components (attributes and roles) that can be partitioned into two distinct classes, each of them describing two types of objects that are related in a one-to-one way, but that are distinct.

De-optimisation technique

Split these components according to these classes, and define a one-to-one rel-type between the entity types obtained (see 7.2.2).

Example

Figure 12.12 shows how entity type A has been decomposed into entity types A and A', each collecting a subset of its components.

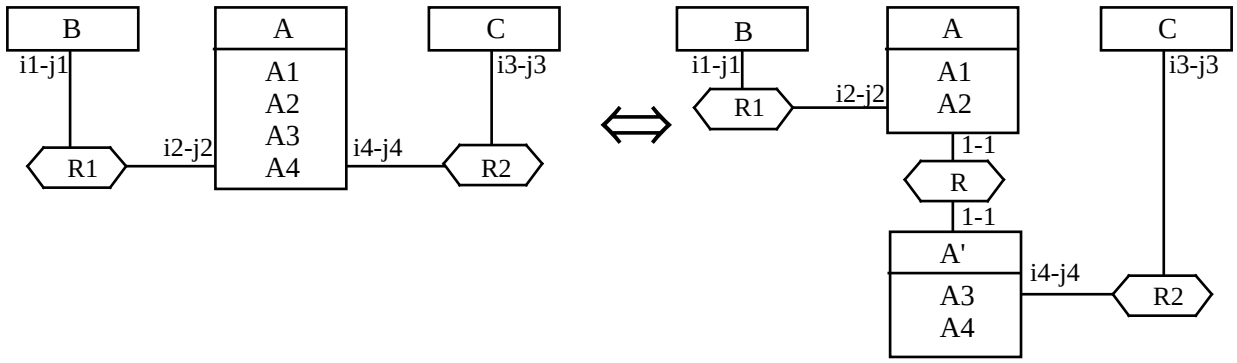


Figure 12.12 *- Splitting of an entity type according to a partition of its components.*

12.4 UNTRANSLATION OF A SCHEMA

This is the process that is the most dependent on the DMS. Consequently, the transformations will be classified according to the most popular DMS, namely standard (i.e. COBOL) file manager, Relational DBMS, CODASYL DBMS, TOTAL/IMAGE DBMS and IMS DBMS. Not all the possible, nor even all the useful transformations can be described in this volume (see technical annex instead). Therefore some of the most needed techniques will be discussed. In addition transformations used for recovering some DBMS schemas can be used for other models as well.

The reader will probably be surprised that the processing of some constructs have been ignored in this section. Such is the case for most many-to-many rel-types, N-ary (N > 2) rel-types, *attribute* entity-types and IS-A hierarchies, that are generally considered in most proposals. In fact these problems have been discarded since they are common to most DMS and there is no general agreement on whether they must be recovered or not. Therefore they will be addressed in the `CONCEPTUAL NORMALIZATION` process instead.

Most transformations that will be presented can be read from left to right, in which case they provide reverse engineering (untranslation) techniques, and from right to left, in which case they represent translation design techniques.

12.4.1 COBOL file structures untranslation

The COBOL data model imposes few constraints on **attribute** structures. The most important one concerns multivalued attributes, which can be represented through *list attributes* only. However, some programmers adopt relational-oriented representations such as *concatenation* or *instantiation*; processing these patterns is proposed in figures 12.16 and 12.17. In addition, optional attributes are not explicitly represented except as multivalued attributes (`. . occurs 1`). This being said, attribute representation will be ignored.

The absence of explicit **rel-type** representation is a more challenging constraint that can be cope with through techniques such as the following. The most popular representation is through foreign keys[1], that can be multivalued (Figure 12.13). The referential constraint is a precondition that can prove difficult to assert since the DMS ignores it. Detecting it will require a careful analysis of the procedural sections of the programs or of the screen definitions. A useful evidence is that many foreign keys are secondary indexes (`alternate keys`) as well.

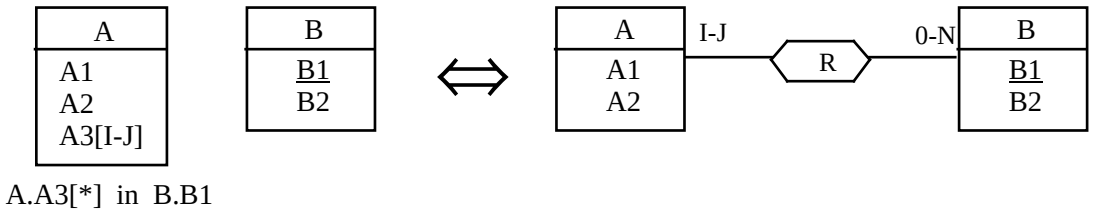


Figure 12.13 *- Representation of a foreign key (A3) by a rel-type.*

Another technique consists in implementing a one-to-many rel-type R between entity types A and B by integrating B entities as instances of a multivalued, compound, attribute of A. Recovering the origin constructs R and B can be done by applying transformation 12.14, where attribute B (possibly compound) of A is transformed into entity type B.

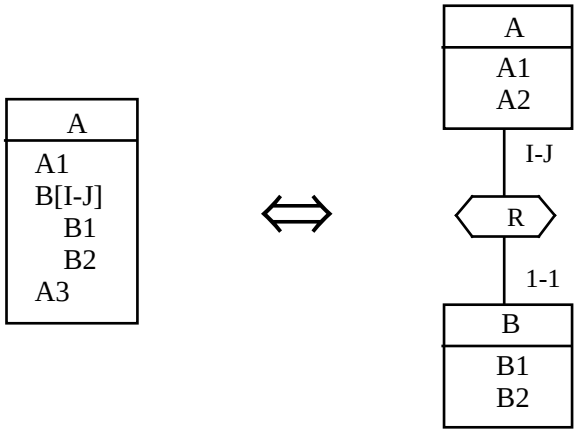


Figure 12.14 *- Representation a compound, multivalued, attribute by an entity type.*

Finally, a one-to-many rel-type can be represented by a sorted multi-record-type sequential or indexed sequential file (Figure 12.15). The keys are structured in such a way that an A

---

[1] Representing a many-to-many rel-type by a relationship record type is not explicitly mentioned. Indeed, this structure can be recovered in two steps, first replace the two foreign keys by one-to-many rel-types, as proposed in Figure 12.19, then apply a conceptual restructuration such as proposed in Figure 12.27 if needed.

instance is followed by its associated B instances in the file sequence. The transformation is not reversible[2] unless a referential constraint from B.B1.B11 to A.A1.A11 can be proved, for instance by file contents examination. However, this physical pattern is sufficiently frequent to make its rel-type origin strongly probable.

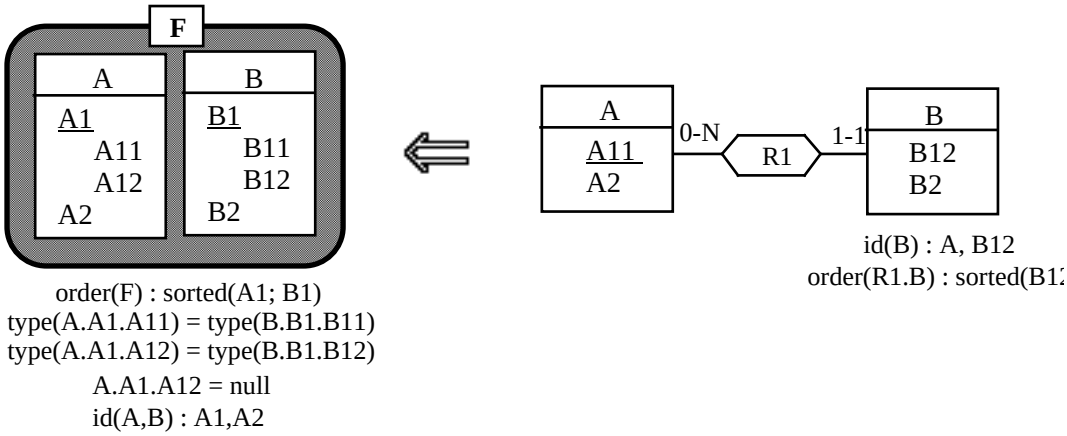


Figure 12.15 *- Representation of a multi-record-type file by a rel-type.*

12.4.2 RELATIONAL schema untranslation

As far as data structures are concerned, the relational model is particularly poor : single-valued and atomic attributes, no rel-types. Therefore, the main problem is to detect representations of multivalued attributes, compound attributes and rel-types.

A **multivalued attribute** can be represented by a distinct table including a foreign key referencing the main table. This pattern is processed by first resolving the foreign key (Figure 12.19), then by conceptual restructuring (Figure 12.14, reverse or 12.22). Two other representation techniques are also frequent, though less elegant, namely *instantiation* and *concatenation* . We will discuss both of them.

The trace of an *instantiation transformation* can be detected by a structure of serial attributes, i.e. a sequence of attributes with the same type and length, and whose names present syntactical (EXP1, EXP2, etc) or semantical (JANUARY, FEBRUARY, etc) similarities. Figure 12.16 illustrates a simple transformation that is adequate for syntactical name similarity. When the uniqueness of their values can be proved, A2 should be declared a *multivalued attribute*.

---

[2] Hence the direction of the transformation symbol, that expresses that the right-side schema can be transformed in the left-side one, but that the converse is not *quite* reversible.

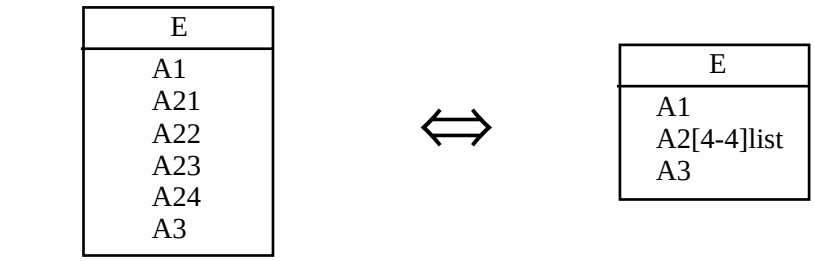


Figure 12.16 *- Representation of homogeneous serial attributes by a multivalued list attribute.*

The *concatenation representation* of a multivalued attribute consists in replacing the set of values by their concatenation, expressed as a single-valued attribute. Its domain appears as possibly made of a repeated simple domain (Figure 12.17). Its detection generally requires the analysis of the procedural code, of the screens layouts or of the table contents.

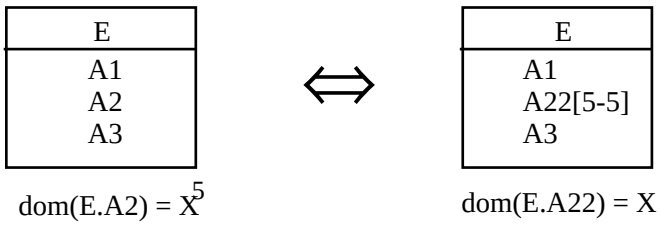


Figure 12.17 *- Representation of a concatenated attribute by a multivalued attribute.*

A **compound attribute** can be represented by concatenation (in a way similar to Figure 12.16), by attribute extraction (Figures 12.14), or by ungrouping, to mention the most frequent techniques. Ungrouping can be detected by the presence of a sequence of heterogeneous attributes whose names suggest a semantic correlation, for instance through a common prefix. Recovering this gouping is straightforward (Figure 12.18).

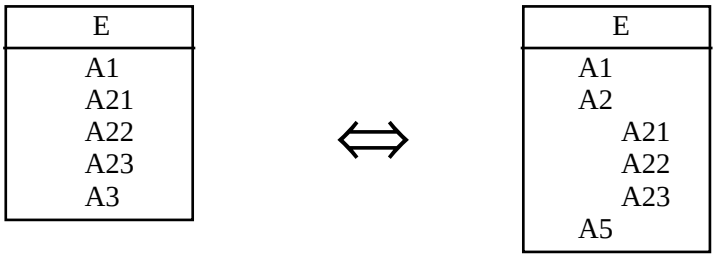


Figure 12.18 *- Representation of heterogeneous serial attributes by a compound attribute.*

Most **one-to-many rel-types** (and one-to-one as well) are represented by single-valued foreign keys (Figure 12.19). Some DBMS provide an explicit representation of them

through the `foreign key` clause. Their validation and management through `check` clauses and `trigger` mechanisms are less straightforward but still easy to detect. Sometimes, there is no declarative hints, and the situation is similar to that of COBOL files. For instance, a common heuristics is that many foreign keys are supported by indexes.

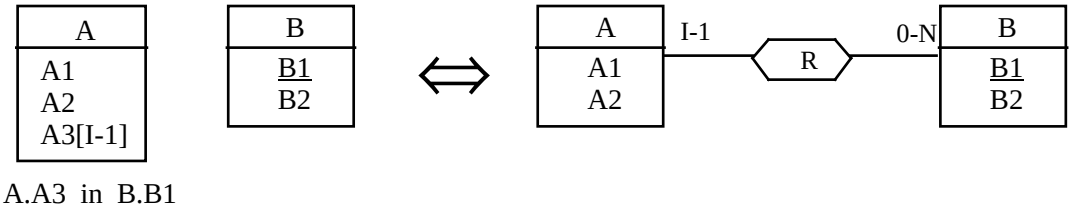


Figure 12.19 *- Representation of a single-valued foreign key by a one-to-many rel-type.*

12.4.3 CODASYL schema untranslation

Among the logical models considered in this paper, the CODASYL DBTG model is the closest to the ER model [PERRON,81] [ELMASRI,93]. As far as conceptual structures are concerned, the main restrictions apply on rel-types (one-to-many and non-recursive) and on identifiers (one absolute id through `location calc`; one relative id per rel-type through `duplicates not` in the `member` clause). Therefore, non-binary rel-types, many-to-many rel-types, one-to-one rel-types, recursive rel-types, secondary absolute identifiers, identifiers with more than one role, have been transformed.

Non-binary and **many-to-many rel-types** will be considered as the target of conceptual restructuration (Figures 12.26 and 12.27).

A **one-to-one rel-type** is implemented either by a one-to-many rel-type, or by a foreign key. Evidence of the first technique can be found through procedural code analysis and data analysis. Processing the second technique is similar to what has been suggested for COBOL and relational models.

A **recursive rel-type** can be represented by an entity type and two one-to-many or one-to-one rel-types. Recovering such a rel-type falls in the conceptual restructuring techniques (Figure 12.28). It can also be represented by a foreign key, as for COBOL and relational models (Figures 12.13 and 12.19).

An entity type with **K all-attributes identifiers** will be inserted into (K-1) SYSTEM rel-types, i.e. rel-types whose (0-N) role is played by the SYSTEM entity type. Each of the (K-1) secondary ids is declared local within one of each such SYSTEM rel-type. In such a situation, and if the SYSTEM rel-type supports no other properties, it is sufficient to discard this set type (Figure 12.20). Another technique consists in extracting the attributes of the id to transform them into an entity type, linked to the main entity type through a one-to-one rel-type. This technique can be detected as an *attribute entity type* (Figures 12.29).

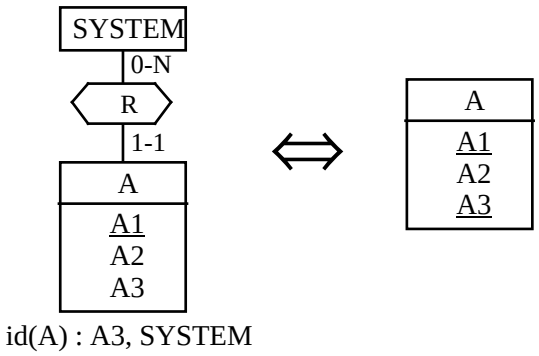


Figure 12.20 *- Recovering an secondary identifier.*

External and mixed identifiers that include **more than one role components** cannot be expressed as such. Either this identifier is discarded, and processed by procedural sections, or all the role components but one are replaced by foreign keys. The latter technique can be reversed as proposed in Figure 12.21. Elicitating the referential constraint requires procedural text or database contents analysis. The schema may keep the source rel-type, according to the principles of *non-information bearing sets* as proposed in the 70's [METAXIDES,75]. In such situations, some DBMS offer a trick (an option of the *set selection* clause) through which the referential constraint is automatically maintained, at least at insert time.

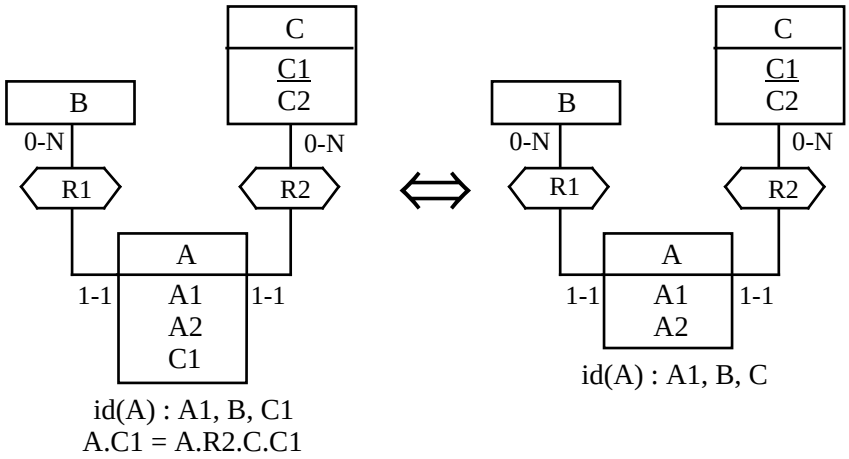


Figure 12.21 *- Recovering a complex identifier (with more than one role component).*

IDS-1 (Honeywell-Bull), that uses a popular variant of the CODASYL model (in fact it is the pre-CODASYL model), requires similar rules, with additional constraints. The same can be said of MDBS-3 and 4 (MDBS) and SIBAS (Norsk-Data). ADABAS (Software AG) structures are often assimilated to CODASYL model [TRICHRITZIS,77]. However, this similarity is rather loose, and this model requires specific translation rules that are out of the scope of this paper.

12.4.4 TOTAL/IMAGE schema untranslation

The TOTAL DBMS (CINCOM), and its clone IMAGE (HP), propose a logical model that is generally classified as network [TRICHRITZIS,77]. However, it seems to fit best as a variant of the hierarchical model[3] as far as design techiques are concerned. This model offers two kinds of entity types, namely the **parent** entity types (the technical name is *master data set*), and the **child** entity types (*variable entry data set*). In addition one-to-many rel-types can be defined between entity types; each rel-type defines access-paths from the parent entities and chidren entities. A parent has single-valued, mandatory and atomic attributes, one of them being its identifier and access key; it can be origin (*one* side) of a rel-type. A child is the target (*many* side) of at least one rel-type. A child entity is the target of at least one rel-type instance (the others can be optional). Among its attributes, there is a copy of the identifier value of each of its parent. These copies behave like redundant *foreign keys* that allow for accessing the parent entities. A child has no identifier. A TOTAL/IMAGE schema is a two-level hierarchy - sometimes called a *shallow* structure - in which level 1 comprises parents only while level 2 is made of children only.

In this model, the problems that occur when translating ER to TOTAL/IMAGE are numerous : expressing complex attributes, non-functional rel-types, one-to-one rel-types, recursive rel-types, parents that are target of some rel-types, children that are source of some rel-types (and in particular hierarchies with more than two levels and non-hierarchical schemas), entity types with more than one identifier or with secondary access keys (IMAGE has a feature for the latter), just to mention the most important. Systematic translation of some of these constructs has been proposed in [HAINAUT,81].

In reverse engineering TOTAL/IMAGE schemas, the redundant foreign keys are detected without problem since they are explicitly declared; they can be discarded without loss. **Compound attributes** are most often processed the same way as in relational schemas. A **multivalued attribute** can be detected either the same way as in relational schemas or as a single-attribute, single-parent, child entity type (figure 12.22).

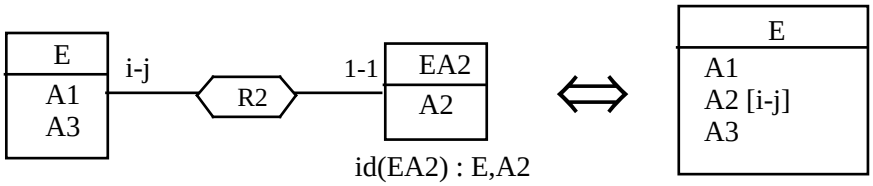


Figure 12.22 *- Recovering a multivalued attribute. If the id(EA2) precondition can be dropped, then A2 is a bag attribute.*

One-to-one rel-types can be processed as in CODASYL schemas. Non-compliant **one-to-many rel-types**, for instance recursive rel-types or rel-types between two parent entity

---

[3] Provided we distinguish the concepts of *hierarchy* and *arborescence*, that are often confused in the DB literature. Considering one-to-many rel-types only, interpreting them as directed arcs (from *one* to *many*) and interpreting entity types as nodes, a schema is said *hierarchical* if its graph has no circuits, it is an *arborescence* if it is hierarchical and if each node has at most one incident arc.

types, are most often expressed as relationship entity types (figure 12.23). They can be recovered provided R1 can be proved one-to-one, by analyzing either the procedural code of the programs or the database contents. Note that some one-to-many rel-types may be expressed implicitely as foreign keys, just like in relational schemas, and are therefore more difficult to detect. As before, recovering many-to-many or higher-degree rel-types is considered as of conceptual improvement concern.

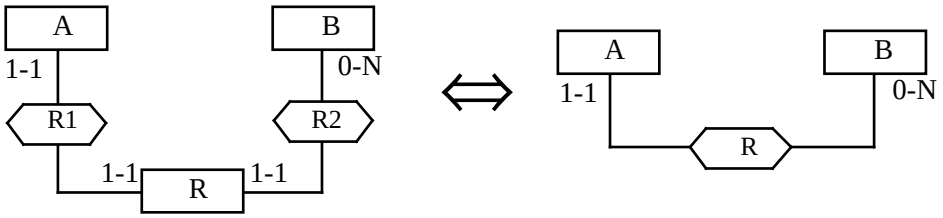


Figure 12.23 *- Recovering a one-to-many rel-type.*

The *2-level hierarchy* constraint implies eliminating (1) **non-hierarchical constructs**, such as circuits, and (2) **deep hierarchies**. Technique 12.23 (reverse) is often used to move A one level up w.r.t. B. Conversely, interpreting child entity type R as a one-to-many rel-type between A and B will automatically recover the origin B-A hierarchy. However, other techniques can be used, such as that which is described in figure 12.24. It requires detecting the inclusion constraint that states that any (`A,B`) instance obtained from one `D` entity must identify a `C` entity (the notation `S1 o S2` expresses the relational composition of `S1` and `S2`). One-to-many rel-type elimination through foreign key and entity type merging techniques are also used.

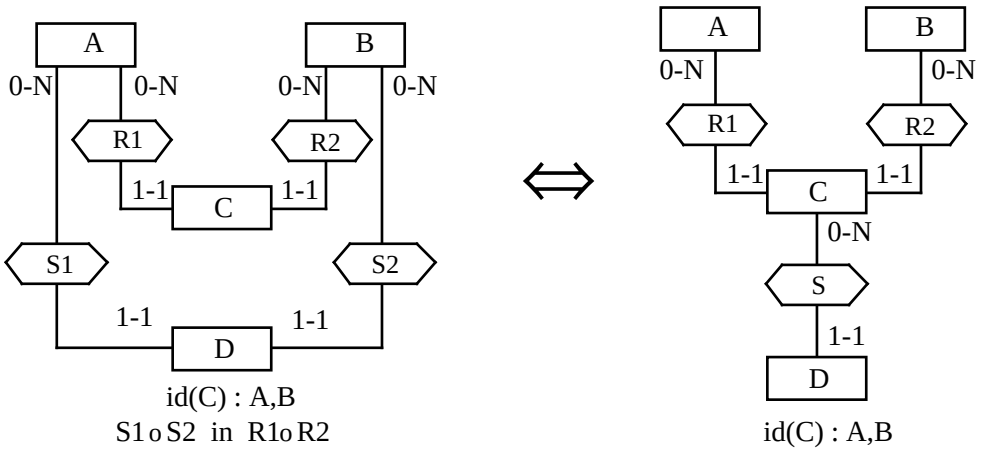


Figure 12.24 *- Recovering a 3-level hierarchy from a 2-level hierarchy.*

12.4.5 IMS schema untranslation

The IMS (also improperly called DL/1) model proposes to structure a schema as a collection of entity types (called *segment types*) linked by *one-to-many* rel-types, that fall into two classes, physical and logical [ELMASRI,93]. The *one* side of a rel-type is a physical or logical parent while the *many* side is a physical or logical child.

Ignoring logical rel-types, the schema reduces to a forest, i.e. a collection of arborescences (or *physical DB's*). The root of each arborescence is called a root entity type (*root segment*). Each root can have one identifier, that is an access-key and can be a sort key as well. It consists of one attribute. Each rel-type defines an access path, from the parent to the child only. A child entity type can have an identifier made of its parent + one local attribute. This identifier is not an access key. Attributes are mandatory, single-valued and atomic. However, compound attributes can be simulated by defining overlapping attributes through common physical positions.

Most IMS schemas are built with the latter constructs[4]. However, two additional features can be used, namely logical rel-types and secondary indexes. A logical rel-type represents an access-path from the child to the parent. A logical rel-type can be defined between any two entity types provided some constraints are satisfied : an entity type can have only one logical parent, a logical child must be a physical child, a logical child cannot be a logical parent, the physical parent of a logical child cannot be a logical child. When bi-directional access paths are needed, IMS proposes to define two, inverse, logical rel-type structures (the *pairing* technique). A secondary index is an access key based on any attribute of the database, whatever its entity type. They generally support no semantic constructs and will be ignored. Surprisingly enough, logical rel-types and secondary indexes are considered in the IMS world as intimidating constructs, difficult and dangerous to use. Even recent references [GELLER,89], though insisting on their harmlessness, suggest to avoid them whenever possible, for instance by replacing rel-types by foreign keys controlled manually !

These constraints define clearly the problems that will appear when translating an ER schema into an IMS structure : compound and multivalued attributes, entity types with several identifiers, one-to-one, many-to-many or recursive rel-types, circuits, entity types with more than two parents, complex identifiers, etc. Additional problems have to be solved when no logical rel-types are used.

Most of these problems have already been discussed and solved in the TOTAL/IMAGE section, or can be considered as relevant to conceptual improvement (e.g. many-to-many rel-types). We shall concentrate on recovering non-compliant **one-to-many rel-types**. Such a one-to-many rel-type can be transformed, for instance, into a foreign key (manually controlled) as in relational schemas, by merging its entity types (producing a possibly unnormalized structure), or into a child entity type. Recovering the source rel-type from application of latter technique is described in figure 12.23. However, to make the process clearer, processing a typical IMS substructure is depicted in figure 12.25, where entity types F and G have been considered as one-to-many rel-type representations. Once again, the main difficulty is to detect the *one-to-one cardinality* of R5 and R7.

---

4 As well as SYSTEM-2000 databases (MRI / SAS) and FOCUS files (Information Builders)

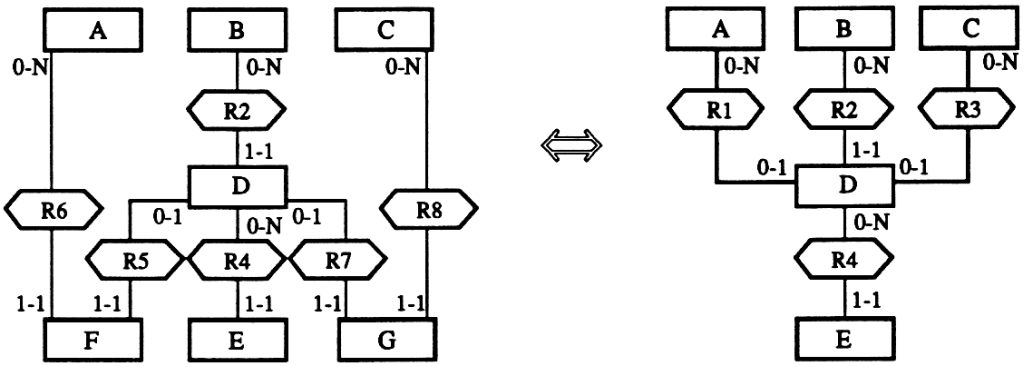


Figure 12.25 *- Recovering one-to-many rel-types.*

Note that duplicate structures resulting from *pairing* can be resolved easily since they are explicitly declared in the DL/1 schema. The redundant logical children can be merged in a preliminary step.

12.5 CONCEPTUAL NORMALIZATION

These transformations are aimed to make higher-level semantic constructs explicit. Whether such expressions are desirable is a matter of methodological standard and of personal taste. For instance, a method that is based on a binary, functional ER model (e.g. the Bachman's model) will generally accept the conceptual schema obtained so far. More powerful models will require the expression of e.g. IS-A relations when relevant. In addition, the final conceptual schema is supposed to be as readable and concise as possible, though these properties basically are subjective.

A in-depth discussion of the quality of a conceptual schema is out of the scope of this manual. The reader can consult [BATINI,92] for instance.

We shall mention some standard transformations that are of interest when refining a conceptual schema. This list is (as usual!) far from complete.

A **relationship entity type**, i.e. an entity type whose aim obviously is to relate two or more entity types, will be transformed into a rel-type (Figure 12.26 to 12.28). This technique typically produces many-to-many and N-ary rel-types, and rel-types with attributes, but can be used for one-to-many rel-ty^pes as well (12.23).

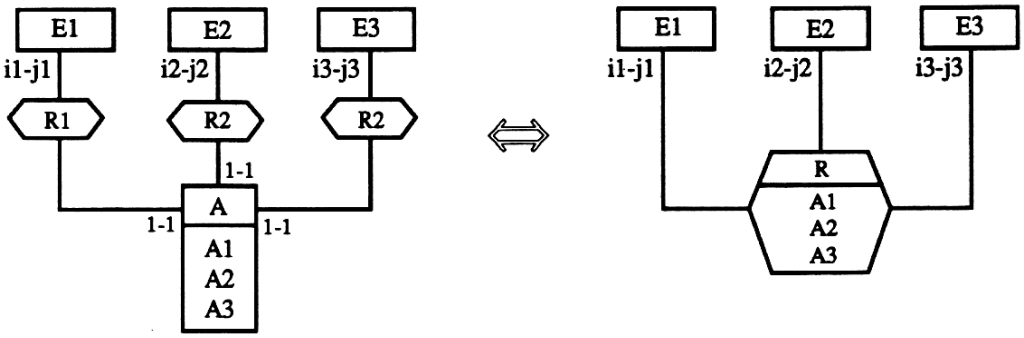
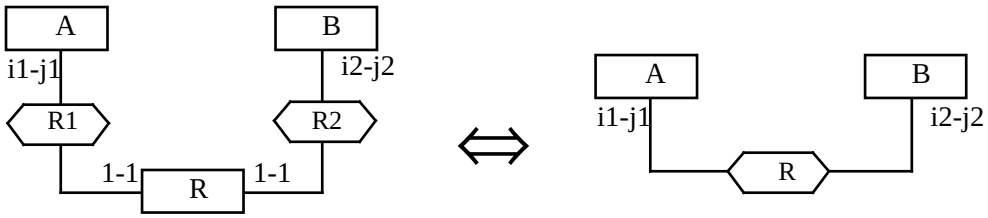
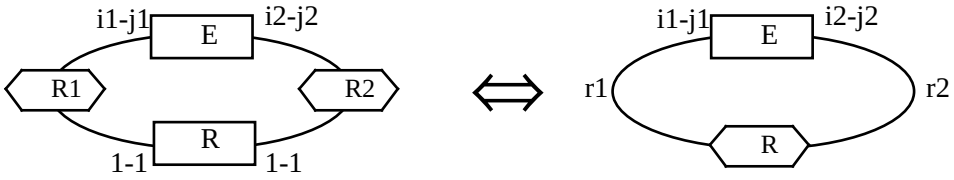


Figure 12.26 *- Transformation of an entity type into a complex rel-type.*



id(R) : A,B

Figure 12.27 *- Transformation of an entity type into a many-to-many rel-type.*



id(R) : R1.E,R2.E

Figure 12.28 *- Transformation of an entity type into a recursive rel-type.*

An **attribute entity type** has one attribute only, and is linked to one other entity type A only. It can be interpreted as nothing more than an attribute of A (Figures 12.29 and 12.30).

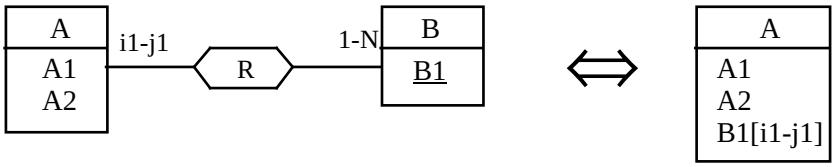


Figure 12.29 *- Transformation of an entity type into an attribute (1).*

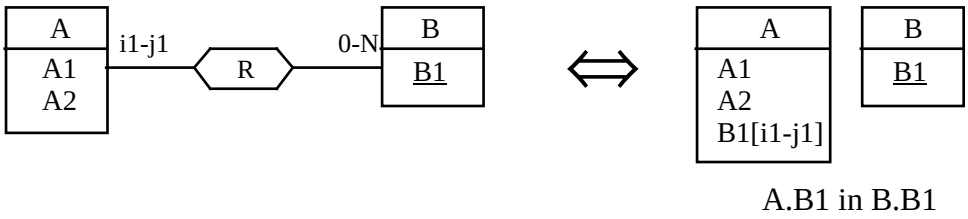


Figure 12.30 *- Transformation of an entity type into an attribute (2).*

A **one-to-one rel-type** may express the connection between fragments A and A' of a unique entity type A (vertical splitting). These fragments can be merged into this entity type (technique 12.31).

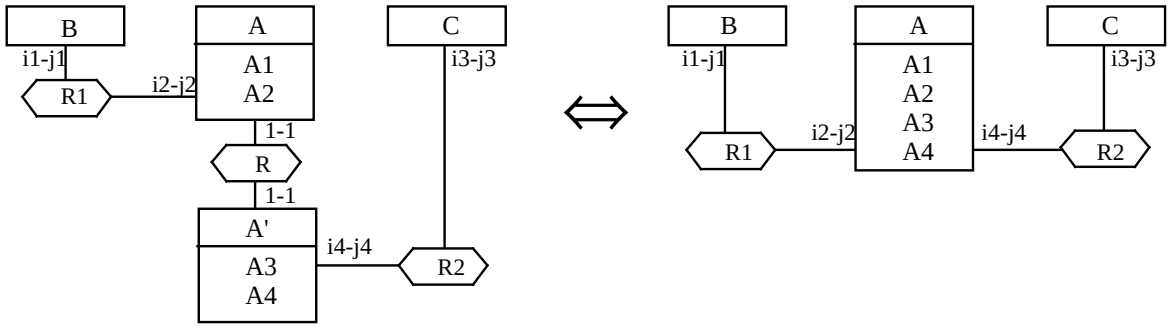


Figure 12.31 *- Merging the components of two entity types.*

An entity type that appears as comprising **too many attributes** and roles can suggest a decomposition into fragments linked by a *one-to-one* rel-type (figure 12.32).

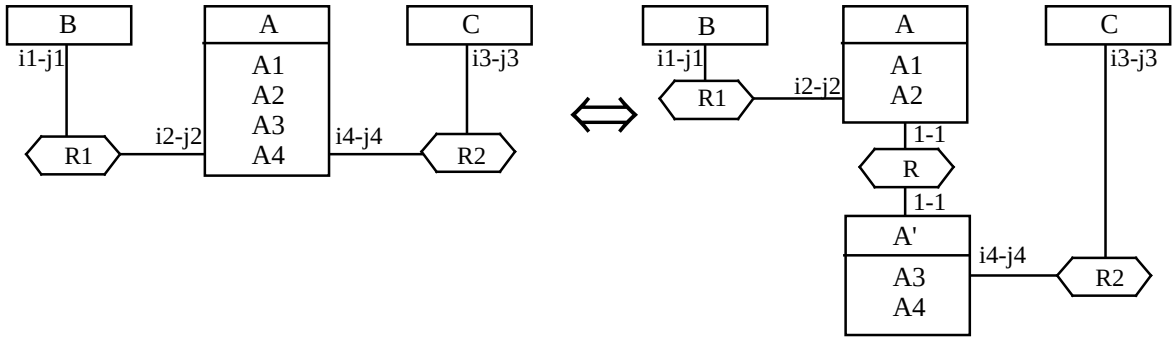


Figure 12.32 *- Splitting the components of two entity types (1).*

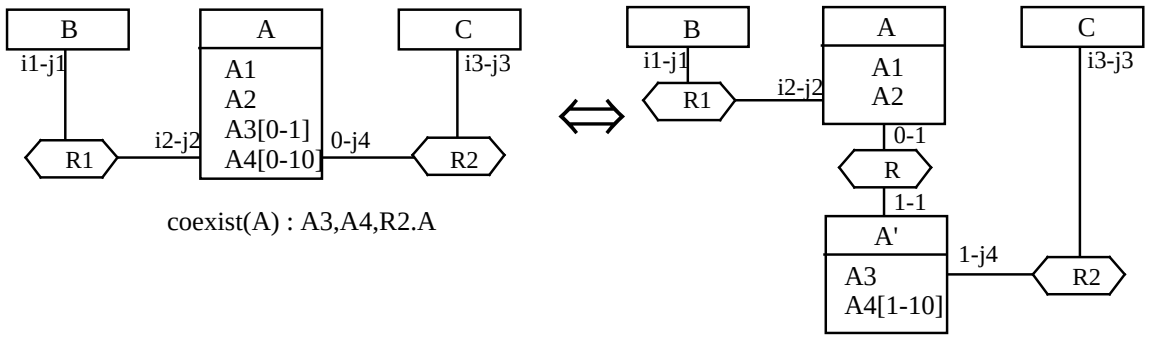


Figure 12.33 *- Splitting the components of two entity types (2).*

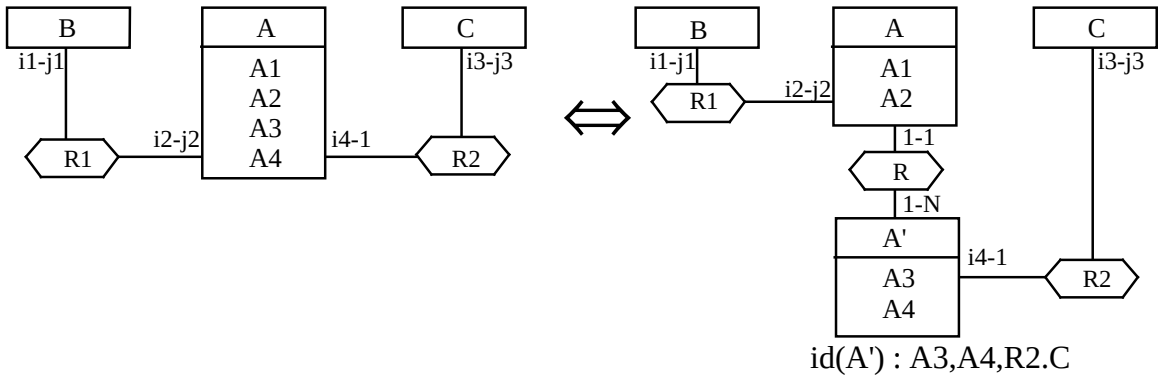


Figure 12.34 *- Splitting the components of two entity types (3).*

A **N-ary rel-type that has a role with cardinality 1-1** can be decomposed into binary, one-to-many rel-types (figures 12.35 and 12.36).

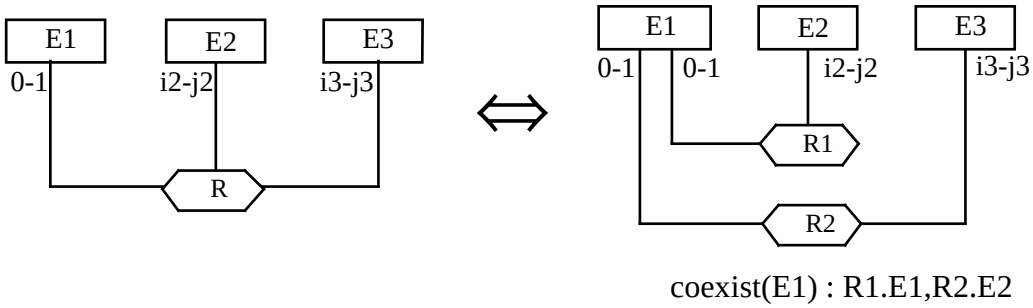


Figure 12.35 *- Decomposition of a rel-type according to a role with cardinality 0-1.*

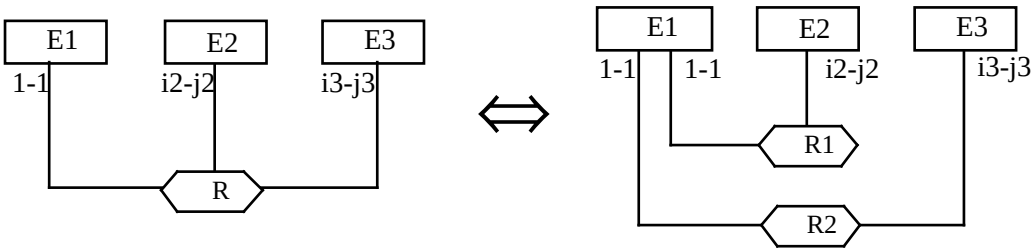


Figure 12.36 *- Decomposition of a rel-type according to a role with cardinality 1-1.*

A collection of entity types that seem to have some **attributes and roles in common** can be made the subtypes of a common supertype that inherits the common characteristics (figure 12.37).

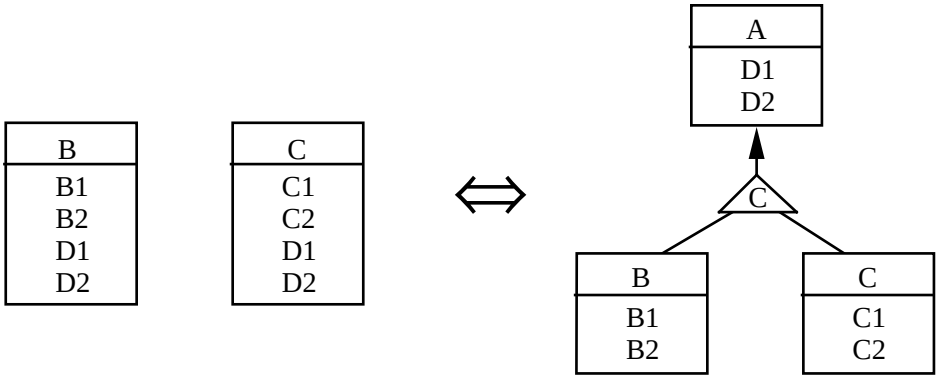


Figure 12.37 *- Extraction of common components from severl entity types.*

One or several **one-to-one rel-types**[5] that concern a common entity type A may also express a specialization relation in which A is the supertype. These rel-types are replaced by IS-A relations figure 12.38).

[5] It is worth noting that a one-to-one rel-type can have three interpretations : true one-to-one rel-type, IS-A relation representation and vertical splitting.

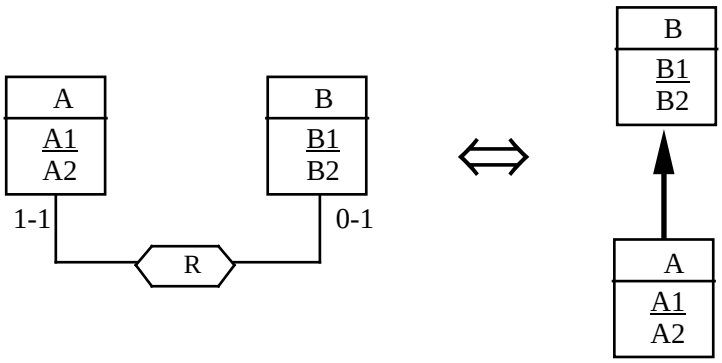


Figure 12.38 *- Expression of a one-to-one rel-type as an IS-A relation.*

An entity type that has **exclusive, optional, subsets of attributes and roles** can be given subtypes, each of them inheriting one of these groups. An entity type that has one group of coexistent, optional attributes and roles can also be examined for such a transformation (figure 12.39).

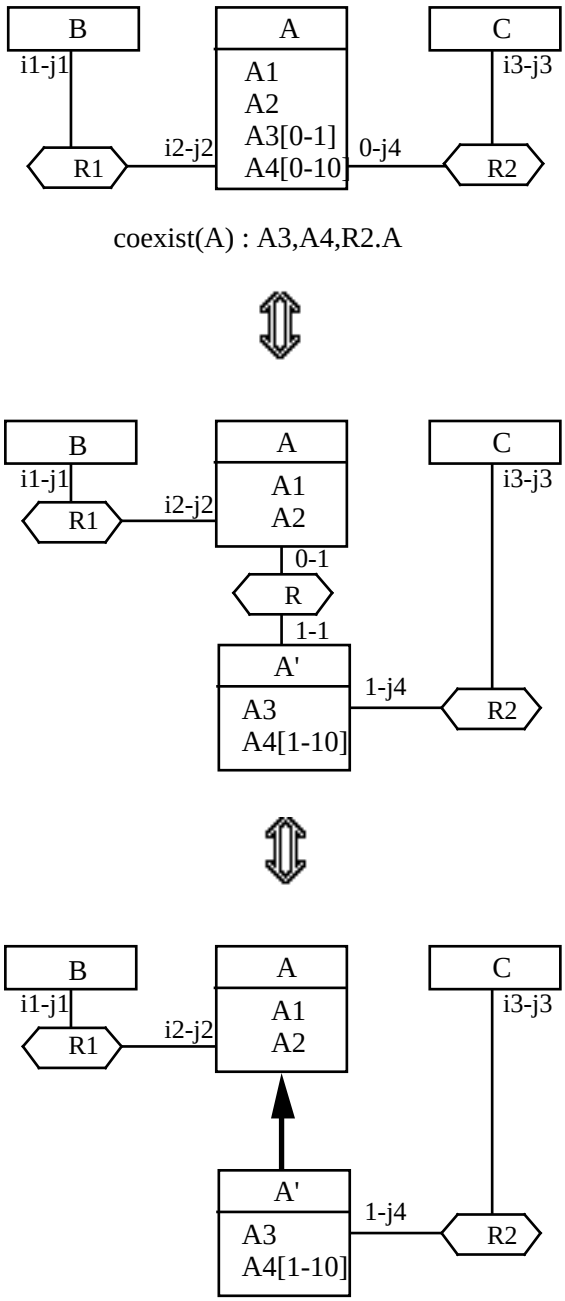


Figure 12.39 *- Expression of a coexistence constraint as a sub/super type structure. This transformation is presented as the composition of vertical splitting (12.33 and expression of a one-to-one rel-type as an IS-A relation (12.38)*

Chapter 13

PHYSICAL/CONCEPTUAL MAPPING

This chapter analyzes why and how it is worthwhile to keep a mapping between the physical and conceptual descriptions of data structures. It defines the concept of object origin. It introduces the notion of version as a mean to register the history of objects.

13.1 THE ISSUE

Here are two examples that illustrate the issue of 'Physical/conceptual description mapping'.

Mapping from physical to conceptual descriptions

The initial data structure extraction usually results in the creation of the basic data structure descriptions handled by the application to reverse engineer. Further analysis provides new kinds of information about these data structures such as specific constraints, more accurate descriptions, etc. However, this 'second' analysis is not necessarily achieved before some work of conceptualization had begun on the previously extracted physical data structure descriptions, whereas the new informations are still defined upon the pure physical data structures. Therefore, the correspondance between these physical descriptions and the conceptual objects into which they have been transformed is needed to pass on the newly acquired information on the conceptualized data structure descriptions.

In Cobol, for instance, the analysis of the 'DATA DIVISION' provides the first basic data structure descriptions. The analysis of the 'PROCEDURE DIVISION' offers additional information such as new descriptions of a data structure on the basis of working-storage

variables with which transfer statements (move, read/write,...) are defined. This information references the physical data structure descriptions extracted on the basis of the analysis of the data division. If these descriptions have already been conceptualized by the reverse engineer via transformation into entity types, integration, or other processes, the newly acquired descriptions must be passed on the corresponding conceptual data structures.

Finally, in a context in which the data administration function is a major client of the reverse engineering process, knowing the meaning (i.e. the conceptual interpretation) of each physical construct is an every-day requirement.

Mapping from conceptual to physical descriptions

Once a final schema is obtained for an application to reverse engineer, it can be used to convert the application into a new system. In this case, to recover the data recorded in the old system, the correspondance between the conceptual data structures defined in the final schema and the basic physical data structure descriptions extracted from the source code is necessary to transfer the data from the old system to the new one.

13.2 A TWOFOLD OBJECTIVE

Maintaining a permanent mapping between the physical and the conceptual descriptions of data structure descriptions meets a twofold objective. One directly relates to the process of reverse engineering itself, the second is concerned with the use of the end result of the reverse engineering.

13.2.1 Throughout the reverse engineering process

The first objective of the physical/conceptual description mapping concerns the reverse engineer in his reasoning processes.

On one hand, all along the reverse engineering process, the reverse engineer needs "historical" information about the objects he has to handle. He will use this information in order to take the right decision about the further conceptualization actions to apply on these objects. In fact, different kinds of physical links may exist between data structures according to their physical definitions (field/record relationship, belonging to the same file, transfer or copy statement, key, clustering organization, ..). The existence of such physical links gives some hints about the existence of conceptual links that could be put in correspondance with them. According to these observations, the reverse engineer can choose specific transformations to apply on the higher-level data structures.

On another hand, let us think about all the activities achieved during the reverse engineering of an application. As already mentionned in section 5.6, these activities of extraction and/or conceptualization may be conducted at any time of the overall reverse engineering process. This methodological approach stresses the need for the reverse

engineer to use historical information corresponding to the physical/conceptual description mapping especially whenever the reverse engineering is partially or momentarily interrupted by regards to a sequential process.

13.2.2 Use of the end result

This second objective has already been introduced in the second example presented in the previous section. When the reverse engineering of an application is completed, the physical/conceptual description mapping may be essential according to the objective(s) of the reverse engineering process : application conversion or data administration for exemple.

In this case, all the data handled by the application program(s) are to be loaded in the corresponding data structures of the newly defined system. This will be easier if an explicit mapping has been kept permanently between the physical descriptions of data structures and their corresponding conceptual data descriptions which definitions have been used to build the new system.

13.3 MAPPING DEFINITION

13.3.1 The Physical/Conceptual Mapping

Given a physical data structure description, the physical/conceptual mapping returns its corresponding conceptual description(s). In other words, in this direction, the mapping answers the following question: 'How has this physical element been conceptualized ?'.

This information may be useful during any reverse engineering reasoning process in which the definition of a physical data structure intervenes. In this case, it can be interesting to check what this physical data structure definition has become through conceptualization, in order to confirm or invalidate the reasoning process.

13.3.2 The Conceptual/Physical Mapping

Given a conceptual data structure description, the conceptual/physical mapping provides the information required to recover the corresponding physical data structures(s). In this way, the mapping answers the question: 'Where does this conceptual data description come from (at the physical level) ?'

Therefore, the conceptual/physical mapping corresponds to the concept of origin of a (conceptual) object.

13.4 VERSION and INTER-VERSION RELATIONSHIP

Let us consider an object whose description is extracted from the source code of an application. It is first regarded as a physical element, belonging to the physical schema of the application. Then, through the reverse engineering process and the various transformations applied on the object, it will progressively become an element of a logical schema and finally of the conceptual schema of the application. These various states of this object can be defined as different versions of the same object.

It is clear that the issue of providing a permanent mapping between physical and conceptual descriptions of an object is to be connected with the one of defining and managing versions of objects.

Anyway, our study of the reverse engineering problem currently points out some characteristics that must lead to specific definitions of a schema element or object and of its different 'versions'. In this matter, the transformational approach of schema elements is fundamental in most of our reasoning processes of reverse engineering. Indeed, since a transformation is a completely formalized mapping between two versions of an object, recording through which transformation (say T) an object has been obtained can provide us with the following information and possibilities :

- the physical/conceptual mapping (i.e. T);
- the conceptual/physical mapping (i.e. T^{-1})
- data conversion
- replaying the reverse engineering processes automatically
- conducting (automatically) a database design starting from the conceptual schema, down to the origin source schemas;

But all this is still a fully open research topic.

13.5 SOME PRACTICAL QUESTIONS

This topic raises a number of other questions that should be solved. Here are some of them.

1. *How detailed and formal sould the mapping be ?*

   This question raises the problem of what must be 'recorded' in the mapping:

   - a strict correspondance between the 'origin' and 'destination' objects (e.g. the transformation with all its parameters),

   - a loose correspondance between the 'origin' and 'destination' objects (e.g. a simple binary relationship between these objects).

2. *Which information must be passed on through processes such as transformation, integration, semantic modification,etc ?*

   or

How 'strict' should the mapping be considered ?

In fact, it seems interesting to keep trace of specific properties of origin objects all through most of these processes. Further analysis of each of theses processes should determined the interest of keeping trace of a more or less accurate mapping. For instance, does any transfer statement defined between two physical data structures remain of interest when considering any of their derived conceptual data descriptions ? Should the physical offset of a field within its record be maintained up to the conceptual level ?

3. As a consequence of the complexity of some transformations applied on objects, the mapping between physical and conceptual descriptions quickly becomes complex as it is composed of several intermediate objects.

   As an example, one sequence of transformations could be:

   a. splitting an entity type A in two (or more) other entity types,

   b. integrating one of the resulting entity types with another entity type B,

   c. normalizing B into entity types C and D

   d. transforming C into a relationship type R.

   Here we see that the retrieval of the physical object corrresponding to the origin of one of a component of R becomes far from trivial.

   There are different ways to manage this complexity. They must be studied and compared in order to choose one of them.

4. In several cases, physical and conceptual elements may have no counterpart.

   As seen in the previous sections, some physical elements (counters, flags, ...) are used to increase performance, to check errors, etc. During the data structure extraction step, many of these constructs are already removed from the data structure descriptions. They can never been accessed at the conceptual level because they have no counterpart in such a schema.

   In the other way, there may also exist conceptual elements with no counterpart at the physical level. In fact, as already mentionned, there is always a loss of information during the transformation of a conceptual schema into a physical schema. This fact implies that no mapping is possible from some conceptual elements, redefined through the reverse engineering process, to any physical element.

Chapter 14

STRATEGIC ASPECTS OF REVERSE ENGINEERING

Discussing strategic aspects has appeared as premature in the current state of our knowledge. However, it is possible to put forward some specific issues that obviously are of strategic nature. This chapter discusses important issues concerning the objectives of reverse engineering, the way(s) in which some specific processes can be carried out, and the origin of the information.

14.1 WHY TO REVERSE ENGINEER A FILE OR A DATABASE ?

The very objective of database and file reverse engineering is to obtain an **up-to-date documentation** of the data structures of an application. As the application data are often an implemented perception of the application domain concepts, the examination of their structures and relations can help a lot in the complete understanding of an existing application and of the application domain itself. An actualized documentation is required to conduct many activities such as application restructuring, maintenance, conversion, extension, integration, new application development or data administration support (see chapter 4).

Therefore, the first key to success is to clearly state the objective of the reverse engineering process.

14.2 WHAT IS THE EXPECTED RESULT ?

The end-products of the database reverse engineering are schemas. Until now, we have based our discussion on the hypothesis that the ultimate goal was to obtain a conceptual schema for the source database. In fact, any intermediate schema can be considered as the result of the process, according to the objective of the organization. Let us examine some outstanding levels of schemas, starting from the physical level, up to the most abstract ones. To simplify the discussion, we have considered a situation in which the source database is made of a set of COBOL files.

- **COBOL description of a file**

  The result is the COBOL declaration of the file. The main problem is to recover the compound field decomposition, and the exact value domains. This result can be inserted in a declaration library, and used through COPY statements in future, or re-engineered programs.

- **COBOL description of a set of files**

  The objective is the same, but applied on a set of files. New problems concern the consistency of names accross the declarations for instance.

- **Augmented COBOL structure of a set of files**

  Here, only the abstract data structures of the files are required, but they should include properties such as referential constraints. This schema clearly is the COBOL-compliant, optimized schema.

- **Simplified COBOL structure of a set of files**

  This schema is a COBOL-compliant description of the files, from which any optimization constructs have been discarded. It represents an ideal simple COBOL structure for the current file contents, that can be converted into this new format.

- **Conceptual view of a program**

  This schema is the conceptual description of the part of the data structures of which the program is aware.

- **Conceptual view of a set of programs**

  This schema is obtained by integrating the conceptual views of each of these programs.

- **The conceptual schema of the application domain**

  This is the conceptual view of the programs, augmented with semantic constructs that have been obtained from external sources.

- **Validation of data structures against a conceptual schema**

  The current data structures are compared against the reference conceptual schema in order to check whether they comply with this schema.

- **Validation of a conceptual schema against the data structures**

  Here, the reference is made of the current data structures. The conceptual schema is compared in order to check whether it is a good description of these structures. Otherwise, this schema will be modified for instance.

- **Augment an existing conceptual schema with concepts used by a program**

  A program must be integrated into, say, an existing application whose conceptual schema is known. Therefore, the latter must be augmented with the specific concepts of the schema of the program.

- **Federated database schema**

  The ultimate goal is to build a global schema (either DMS-compliant in case of homogeneous DMS, or conceptual) that describes all the data structures of a set of existing databases.

14.3 WHAT ARE THE QUALITIES OF THE FINAL SCHEMA ?

According to the multiplicity of objectives when reverse engineering an application, qualities of the final schema may be judged in different ways. Therefore, it seems more adequate to introduce evaluation criteria on the basis of which qualities of a schema may be discussed.

Evaluation criteria

Here is a list of criteria helpful in assessing the qualities of the final schema obtained by reverse engineering an application.

1. *Closest image of the RE'ed application*

   If the objective of reverse engineering implies to recover an accurate image of the application, for application redocumentation or conversion for instance, it is

important that the final schema is an image as close as possible of the RE'ed application.

On the contrary, when the ultimate objective of RE relates to application extension or development for instance, to get the closest image of the RE'ed application is not the ultimate aim and a further work of conceptualization can be done on this first result to improve, correct and extend the schema obtained.

2. *Integration, redundancy reduction, normalization*

   During the global process of reverse engineering, integration, redundancy reduction and normalization are tasks to be achieved in order to build up a final schema which minimizes redundancy and is an integration of all the pertinent descriptions issued from the different file sources of the application.

   Nevertheless, if an application source code introduces the definition of several views specific to different users, the reverse engineer can preserve or disregard the application structure in views. Depending on this choice, the final schema can be divided in accordance with the different views or it can be an integration of all the view definition.

3. *Completeness*

   The final schema of an application must be complete according to the image of the real world that the reverse engineer wants to describe.

   Moreover, when considering the basic data structures defined in the schema, they must include or at least allow the definition of as many as possible integrity constraints that are to be defined.

4. *Semantic enrichment*

   The final schema obtained by reverse engineering an application may be enriched by any semantical information considered as pertinent or significant. Nevertheless, the more a schema is enriched, the less it is close to the genuine image of the application.

5. *Clarity, conciseness, readableness*

   As a medium of communication, the schema of an application must be clear, concise and readable. See section 12.6 for practical strategies and techniques for improving such characteristics.

6. *Compliancy with a methodological standard*

   The final schema of the RE'ed application must be compliant with any existing methodological standard. If so, for any other person than the reverse engineer, the schema is easier to understand since it is positionned at a specific place within the development methodology and thus must meet the corresponding specific features.

14.4 WHEN WILL NAME PROCESSING OCCUR ?

14.4.1 Very early name processing

Name processing at a very early stage means processing before any data structure examination has occurred.

If we relate this to the reverse engineering methodology described in chapter 5 then we can situate very early name processing before going back from the physical schema to the DMS-compliant logical schema.

This is a possibility if we know that the data structures come from a data dictionary (formal or informal, automated or by oral or written corporate standards) and if these structures have been processed before in the course of previous reverse engineering attempts. In that case we know what names to expect and how they should be interpreted and processed.

The advantage of the very early processing is that you can work with meaningful names from the start. The distinction between semantical and nonsemantical structures is clear from the names and the structures that are relevant for the reconstruction of the conceptual schema can be filtered out immediately.

If the data structures have not been interpreted before then name processing at a very early stage is not recommended.

14.4.2 Early name processing

Name processing at an early stage means processing during the data structure extraction step.

This corresponds more or less with the transition from the physical to the logical schema in the proposed reverse engineering methodology (see chapter 5).

In the course of looking for the logical data files, the records and the fields from the source name processing can occur. As the data structures are found their names are processed. As in the first case subsequent reverse engineering sessions will benefit from the more meaningful names.

The early name processing is easy to apply for every application one would like to reverse engineer. As the names appear they may be changed, expanded, truncated, translated and so on.

Early name processing may prove impossible if the interpretation and the semantics of a structure become clear only after the data structure extraction step.

Summary for early and very early name processing :

The advantage of early processing is the fact that the semantics are there from the start. The other stages in the reverse engineering attempt will profit highly from the clear meaning of the data structures.

A disadvantage of early processing is that it may make some techniques impossible in further stages of reverse engineering. Slightly different names or the use of prefix and suffix make sense to implement semantics in an environment like COBOL that provides only poor semantics. Removing these clues at an early stage will make it very difficult if not impossible to recover these semantical structures. This is true for very voluminuous applications where one loses track very quickly about processed structures.

Name processing techniques like translation, truncation, expansion... are less apt to this problem than names with a prefix or a suffix. It is important to establish the meaning of a prefix or a suffix before processing the name.

When a name appears several times with a different prefix or suffix one should be cautious about removing it too soon. The different 'values' of the prefix or suffix may suggest subtypes or other semantical structures.

Example :

* M-SOUSCRIPT-04
    - QVN-04
    - NET-CLI-04
    - VRS-BRUT-04
    - INT-04

* M-RACHAT-02
    - QVN-02
    - MON-IMPOS-02
    - VAL-RACHAT-02
    - TYPE-02

The composed fields M-SOUSCRIPT-04 and M-RACHAT-02 have nothing but suffixed subfields. Source analysis (bot DMS and procedural sections) revealed that a field IND-MVT is decisive for the kind of transaction to be executed. The meaning of this field is 'indicateur du mouvement'. IND-MVT can have a limited set of values; value 02 is linked with selling shares (rachat) while value 04 is linked with subscribing to new shares (souscript). The suffix added to the fields of M-SOUSCRIPT-04 and M-RACHAT-02 is meaningful and contributes to the establishment of a relationship between subtypes of IND-MVT on the one hand, and M-SOUSCRIPT-04 and RACHAT-02 on the other one.

The addition of the suffix in both cases was strong evidence of M-SOUSCRIPT-04 and M-RACHAT-02 being entity types. The fact that both fields are composed is yet evidence but the addition of the common suffix adds to the plausibility.

An early removal of the suffix in this case would have obscured these semantics considerably.

14.4.3 Late name processing

Name processing at a late stage means processing during the data structure conceptualization step. The advantage of late processing can be deduced from the example above. One can fully use the semantics of nonsemantical names to derive conceptual structures. Once the structures are in places the names can be processed.

The disadvantage of course is that highly nonsemantic names will blur the sight on the meaning of the underlying objects.

Removal of meaningful prefix or suffix and string replacement (complete renaming) is preferably delayed until the data structure conceptualization step when they may play a role in reconstructing important conceptual structures.

14.5 WHEN CAN SCHEMA INTEGRATION OCCUR ?

Chapter 10 has presented the different techniques which are useful to integrate several data structures called views, when they have semantic interconnections. Our intention was to propose general tools, which are valid whatever the kind of the structures involved or the moment of their application.

However, that does not mean that no advice can be given on the use of these techniques. There are more or less judicious moments in the reverse engineering to use them. Two situations must be distinguished :

- **integration of physical data structures**, i.e. integration of data structures very close to the original ones which were extracted directly from the source texts. This kind of integration can exploit specific physical informations (length, starting adress, physical file, ...) for correspondence detection, which increases the automatization degree of the integration.

  When data structures redundancies are numerous, this automatization is very interesting and justifies therefore an early integration for two reasons :

  - this early integration will reduce largely the size and the complexity of the specifications for further processing.
  - the data structures are going to be modified by other processes (transformation-enrichment); their physical informations will become less relevant, increasing by this way the complexity of using these informations later.

On the other hand, this kind of integration, when performed early, will be hampered by the fact that the involved structures are less-conceptual; so semantic evaluations are more difficult. The following factors make the structure less understandable (see chapter 12) :

- the structures are DMS-compliant,
- the structures contain optimization and technical constructs,
- the structures are unnormalized,...

- **integration of more semantical structures**. Correspondence detection for these structures can only be based on reference to real-world facts. It is essential to elicit totally the semantics of these structures before going to integrate them (moreover, there are so various possibilities of corresponding structures). The semantics of each concept must be clearly understood.

  So we propose to process this kind of integration in later steps of the reverse engineering.

14.6 HOW TO PROCEED WHEN INTEGRATING SEVERAL SCHEMAS ?

Three general strategies for integrating more than two views are presented in chapter 10, section 10.4 :

- n-ary integration strategy
- binary balanced tree-like strategy
- binary incremental strategy

There are two strategic problems that a user can encounter : the first one is the size of the work, in terms of structures to be analyzed and sometimes to be integrated. the second problem, which results from the first one, is the need of a systematic approach, in order to be sure that no redundancies will remain in the final result.

Two principles should be interesting to help the reader who has to realize an integration process, and more particularly to organize and systematize it :

- try to construct **non-redundant sets of structures**, i.e. sets of structures which contain no intra-redundancies. They will be small at the beginning, then larger. Knowing that there are at least no redundancies in some parts of the specifications can put the user's mind at ease for structuring his work.
- **Incremental enrichment**. A good strategy can be to choose one view as the main one, then to integrate successively other views to that one. This main view contains more and more integrated structures.

  How to choose the main view? It can be the largest one, or the most reliable one (for instance, its source must be reliable : an interview is certainly less reliable than the source text of the main database process, at least from a reverse engineering point of view.).

By combining these two ideas, and using the concept of schema as a unit of work made out of a set of structures, the following method can be proposed for instance :

1. for each schema, suppress the structure redundancies between elements of these schema (if the schema is large, define several subparts and apply these principles on these subparts).
2. choose a schema as the main one, and integrate each other schema with that one.

14.7 WHERE TO FIND THE INFORMATION ?

Though the file and database descriptions and source text programs are the main source of information in the reverse engineering of databases, it is clear that they do not contain some essential information or at least, it is too difficult to retrieve some of them when spread over the program. To accomplish his task, the reverse engineer must identify and find missing information. Who are the final users of the application, who have developed it, is there some existing documentation, data dictionary, screen forms or generated reports, what are the data instances of the data files ? Such questions will lead the reverse engineer to new sources of information which will help him to validate or invalidate initial presumptions, or will give him new research orientations. All these new sources of information can also be precious to the reverse engineer before he starts with the database descriptions or the source texts analysis.

As the exploitation in a reverse engineering context of each source of information is in itself a subject of research, we just intend here to suggest how they can be a useful complement to the reverse engineering of database source texts.

14.7.1 Source texts

The importance of source texts and specific DBMS descriptions has been described in previous sections. The amount of source texts for a medium size application can easily grew up to hundred of thousands lines of code. The problem for the reverse engineer is to be able to manage such an amount of information (see chapter 11, section 11.2).

14.7.2 Data entry forms and reports

The data entry forms and the reports can be helpful to find right names for objects, to select pertinent semantic informations from technical ones, to suggest relations between data descriptions following their spatial presentations... Their deep study could also lead to a partial view of the data structure of the system which could to be integrated or confront with the result of the source code analysis.

14.7.3 Existing documentation

All existing documents on the analyzed application or its domain can often reveal some interest. These documents can consist of the program documentation or comments, programmer notes, service notes or letters dealing with the computerization decision, contracts, ... The problem is about the reliability of such sources of information which may be wrong or obsolete (up-to-date and complete documentation is most often missing when reverse engineering is needed!) Anyway, they can often give good general introduction ideas to the reverse engineering work.

14.7.4 Data dictionaries

Data dictionaries contain data descriptions, version, usage, definition and relation. When application programs copy their data descriptions from dictionaries, the data dictionaries content can help a lot to detect data redundancy, relations between data, data constraints, to retrieve original data definitions and to facilitate schemas integration The examination of existing data dictionary can also suggest extensions to the application schema with concepts related to the data of the schema.

14.7.5 CASE tool

At first glance, the contents of the repository of any CASE tool can be considered as the final product of reverse engineering of the application that this tool has built. Therefore one could consider that the problem is solved.

However, due to the weaknesses of the current CASE technology, the operational system have often been edited manually in order to give it some characteristics that the tool was unable to provide.

In addition, the operational system have inevitably evolved, while the conceptual specifications (i.e. the contents of the repository) have generally been left unchanged.

14.7.6 Data file contents

The data file contents reveal exactly what are the modelized real world facts and objects at a certain time. Data analysis can be very useful to detect integrity constraints, such as identifiers, referential integrity, exclusive constraints, functional dependencies or redundencies. It can also provide essential hints on the value domains and field decomposition.

14.7.7 Developer interview

The most important source of knowledge to reverse engineer an application is the developer himself, provided he still is available, and he remembers the architecture and the details of the application.

14.7.8 User interview

Most generally, the users of the application cannot bring useful information on the file structures, but they can give essential information on the behaviour of the application, on the data that appear on the screen and in the reports. In addition they can provide a reliable description of some aspects of the application domain concerned by the programs.

Chapter 15

A SHORT SQL CASE STUDY

This short case study illustrates the main concepts, processes and reasonings developed in this manual through the reverse engineering of a relational database.

15.1 PRESENTATION

This chapter presents a small size database reverse engineering example that includes the analysis of both declarative and procedural source code. The context is that of a small technical library[1]. The application is undocumented, but is fairly well structured. In particular, the basic data structures are rather clear, thanks to, a.o., the choice of explicit names. In addition, the source DMS is some modern form of Relational DBMS that allows for explicit declaration of primary and foreign keys (though this possibility may not be exploited in some cases).

To make the development more readable and more general, we have decided to express the source code in SQL for the database data structures, and in a sort of pseudo-code for the procedural sections. The extension to, say, *COBOL-SQL* or *PL/1-Standard files* expressions is fairly straighforward.

This case study has been developed in such a way that some of the most important aspects of the reverse engineering process are illustrated. For instance, primary keys are not

---

[1] The same example has been used to develop a database design case study that is reported in,

Hainaut, J-L., *Conception de bases de données - Etude de cas*, Technical report, Lausanne, April, 1992, EPFL & F.U.N.D.P., 80p.

always explicitly declared, non standard relational translation rules and optimization restructuration techniques have been used, some integrity constraints have been translated into procedural code.

However, other important problems that could have occured in practice have not been illustrated. Let us only mention some of them :

- absence of source code for the global schema, as in standard file managers,
- absence of declared foreign keys, as in older RDBMS and in standard file managers,
- more intensive use of user views to refine the global schema,
- low-information names,
- synonyms and homonyms,
- erroneous and conflicting source information,
- complex view integration,
- more complex optimization techniques,
- use of inductive reasonings based on database contents,
- partial or corrupted source code,
- presence of non semantic constructs, dead structures, dirty programming tricks,
- etc.

This document presents the case through its declarative (in SQL DDL) and procedural (in some SQL-DML pseudo-code) source code, and an example of resolution.

15.2 THE SOURCE CODE TEXTS

15.2.1 The SQL schema

```
create table AUTEUR (
                CODE_AUTEUR integer not null ,
                NOM varchar(30) not null ,
                PRENOM varchar(20) ,
                DATE_NAISSANCE char(6) ,
                ORIGINE varchar(30),
                primary key (CODE_AUTEUR)
            )

create table EMPRUNTEUR (
```

```
                MATRICULE char(10) not null ,
                NOM varchar(30) not null ,
                PRENOM varchar(20) not null ,
                FONCTION varchar(20) not null ,
                AD_SOCIETE char(20) not null ,
                AD_VOIE varchar(40) not null ,
                AD_CODE_POSTAL integer not null ,
                AD_VILLE varchar(30) not null ,
                TELEPHONE1 varchar(20) not null,
                TELEPHONE2 varchar(20),
                TELEPHONE3 varchar(20),
                TELEPHONE4 varchar(20),
                TELEPHONE5 varchar(20),
                REPONDANT char(10),
                primary key (MATRICULE),
                foreign key (REPONDANT) references EMPRUNTEUR
            )

create table EMP_CLOTURE (
                NUMERO_OUVRAGE char(10) not null ,
                NUMERO_D_ORDRE varchar(2) not null ,
                MAT_EMPRUNTEUR char(10) not null ,
                CODE_PROJET char(3) not null,
                DATE_DEBUT char(6) not null ,
                DATE_FIN char(6) not null ,
                foreign key (NUMERO_OUVRAGE,NUMERO_D_ORDRE)
                        references EXEMPLAIRE,
                foreign key (MAT_EMPRUNTEUR) references EMPRUNTEUR,
                foreign key (CODE_PROJET) references PROJET
            )

create table EXEMPLAIRE (
                NUMERO_OUVRAGE char(10) not null,
                NUMERO_D_ORDRE varchar(2) not null ,
```

```
                TITRE char(50) not null ,
                DATE_D_ACQUISITION char(6) not null ,
                LOCALISATION char(6) not null ,
                NOMBRE_DE_LIVRES smallint not null ,
                ETAT_LIVRE char(1) not null ,
                COMMENTAIRE_ETAT varchar(80) ,
                MAT_EMPRUNTEUR char(10),
                CODE_PROJET char(3),
                DATE_EMPRUNT char(6),
                primary key (NUMERO_OUVRAGE,NUMERO_D_ORDRE),
                foreign key (NUMERO_OUVRAGE) references OUVRAGE
                foreign key (MAT_EMPRUNTEUR) references EMPRUNTEUR,
                foreign key (CODE_PROJET) references PROJET
            )

create table MOTS_CLES_D_OUV (
                MOT_CLE varchar(15) not null ,
                NUMERO_OUVRAGE char(10) not null,
                foreign key (NUMERO_OUVRAGE) references OUVRAGE
            )

create table ECRIT (
                CODE_AUTEUR integer not null,
                NUMERO_OUVRAGE char(10) not null ,
                foreign key (NUMERO_OUVRAGE) references OUVRAGE,
                foreign key (CODE_AUTEUR) references AUTEUR
            )

create table OUVRAGE (
                NUMERO char(10) not null ,
                TITRE char(50) not null ,
                EDITEUR varchar(30) not null ,
                DATE_PREM_PARUTION char(6) not null ,
                primary key (NUMERO)
```

```
            )

create table COMPL_OUVRAGE (
                NUMERO_OUVRAGE char(10) not null ,
                NOTE_PRESENTATION long varchar,
                foreign key (NUMERO_OUVRAGE) references OUVRAGE
            )

create table PROJET (
                CODE char(3) not null ,
                NOM varchar(30) not null,
                primary key (CODE)
            )

create table REF_BIBLIOGRAPH (
                NUMERO_ORIGINE char(10) not null ,
                NUMERO_REFERENCE char(10) not null,
                foreign key (NUMERO_ORIGINE) references OUVRAGE,
                foreign key (NUMERO_REFERENCE) references OUVRAGE
            )


create unique index AUTndx1 on AUTEUR (CODE_AUTEUR)

create unique index EMPndx1 on EMPRUNTEUR (MATRICULE)

create unique index EM1ndx1 on EMP_CLOTURE
                (NUMERO_OUVRAGE,NUMERO_D_ORDRE,DATE_DEBUT)

create unique index EXndx1 on EXEMPLAIRE (NUMERO_OUVRAGE,NUMERO_D_ORDRE)

create unique index XMOndx1 on MOTS_CLES_D_OUV (MOT_CLE,NUMERO_OUVRAGE)
create index XMOndx2 on MOTS_CLES_D_OUV  (NUMERO_OUVRAGE)
```

```
create unique index OVAndx1 on ECRIT (NUMERO_OUVRAGE,CODE_AUTEUR)
create index OVAndx3 on ECRIT  (CODE_AUTEUR)

create unique index OUVndx1 on OUVRAGE (NUMERO)

create unique index COndx1 on COMPL_OUVRAGE(NUMERO_OUVRAGE)

create unique index PRJndx1 on PROJET (CODE)

create unique index PRJndx2 on PROJET (NOM)

create unique index REFndx1 on REF_BIBLIOGRAPH
                 (NUMERO_ORIGINE,NUMERO_REFERENCE)
create index REFndx3 on REF_BIBLIOGRAPH  (NUMERO_REFERENCE)
```

15.2.2 Program CHECK-BIBLIO

```
program CHECK_BIBLIO
declare section
   CODE_AUT : integer;
   ERROR_FILE : file of integer;

procedure section
   open file ERROR_FILE;
   declare A cursor for
      select CODE_AUTEUR
      from AUTEUR
      where CODE_AUTEUR not in (select CODE_AUTEUR from ECRIT);
   open A;
   fetch A into :CODE_AUT;
   while (SQLCODE = 0) do
      write CODE_AUT into ERROR-FILE;
```

```
      fetch A into :CODE_AUT;
   end-while;
   close A;
   close ERROR_FILE;
end-program
```

15.2.3 Program BIBLIO-MANAGEMENT

```
program BIBLIO_MANAGEMENT

module DISPLAY_EXEMPLAIRE (NOU:char[10]; NOR:char[2])
declare section
   IN_EX : record
             NOUV char[10];
             NORD char[2];
             TITRE char[50];
             DACQ char[6];
             LOCAL char[7];
             NBL smallint;
             ETAT char[1];
             COMETAT char[80];
           end-record;

   OUT_EX : record
             NOUV char[10];
             NORD char[2];
             TITRE char[50];
             DACQ char[6];
             ETAGE smallint;
             TRAVEE char[2];
             RAYON char[3];
             NBL smallint;
```

```
            ETAT char[1];
            COMETAT char[80];
          end-record;

procedure section
   select
      TITRE,DATE_D_ACQUISITION,LOCALISATION,
      NOMBRE_DE_LIVRES,ETAT_LIVRE,COMMENTAIRE_ETAT
   into
      :IN_EX.TITRE,:IN_EX.DACQ,:IN_EX.LOCAL,
      :IN_EX.NBL,:IN_EX.ETAT,:IN_EX.COMETAT,
   from EXEMPLAIRE
   where NUMERO_OUVRAGE = NOU and NUMERO_D_ORDRE = NOR;
   IN_EX.NOUV := NOU;
   IN_EX := NOR;
   if SQLCODE = 0 then
      OUT_EX := IN_EX;
      display_EX(OUT_EX);
   else
      ...
   end-if;
   ...
end-module DISPLAY_EXEMPLAIRE


module CREATE_EXEMPLAIRE (NOU:char[10])
declare section
   TITRE : char[50];
   IN_EX : record
            NORD char[2];
            DACQ char[6];
            ETAGE smallint;
            TRAVEE char[2];
            RAYON char[3];
```

```
            NBL smallint;
            ETAT char[1];
          end-record;
   OUT_EX : record
            NORD char[2];
            DACQ char[6];
            LOCAL char[6];
            NBL smallint;
            ETAT char[1];
          end-record;

procedure
   input_EX(IN_EX);
   OUT_EX := IN_EX;
   select TITRE into :TITRE from OUVRAGE where NUMERO = :NOU;
   insert into EXEMPLAIRE (NUMERO_OUVRAGE,NUMERO_D_ORDRE,TITRE,
          DATE_D_ACQUISITION,LOCALISATION,NOMBRE_DE_LIVRES,ETAT_LIVRE)
          values (:NOU,:IN_EX.NORD,:TITRE,
                  :OUT_EX.DACQ,:OUT_EX.LOCAL,:OUT_EX.NBL,:OUT_EX.ETAT);

end-module CREATE_EXEMPLAIRE


module UPDATE_EXEMPLAIRE (NOU:char[10]; NOR:char[2])

declare section
   UPDATE : boolean;
   USER_EX : record
            DACQ char[6];
            ETAGE smallint;
            TRAVEE char[2];
            RAYON char[3];
            NBL smallint;
            ETAT char[1];
```

```
            end-record;
    DB_EX : record
              DACQ char[6];
              LOCAL char[7];
              NBL smallint;
              ETAT char[1];
            end-record;

procedure section
    select
       DATE_D_ACQUISITION,LOCALISATION,
       NOMBRE_DE_LIVRES,ETAT_LIVRE
    into
       :DB_EX.DACQ,:DB_EX.LOCAL,
       :DB_EX.NBL,:DB_EX.ETAT
    from EXEMPLAIRE
    where NUMERO_OUVRAGE = :NOU and NUMERO_D_ORDRE = :NOR;
    USER_EX := DB_EX;
    update_data_EX(USER_EX,UPDATE)
    if UPDATE then
       DB_EX := USER_EX;
       update EXEMPLAIRE
       set DATE_D_ACQUISITION = :DB_EX.DACQ,
           LOCALISATION = :DB_EX.LOCAL,
           NOMBRE_DE_LIVRES = :DB_EX.NBL,
           ETAT_LIVRE = :DB_EX.ETAT
       where NUMERO_OUVRAGE = :NOU and NUMERO_D_ORDRE = :NOR;
    ...
end-module UPDATE_EXEMPLAIRE
```

```
module DELETE_EMPRUNT (NOU:char[10]; NOR:char[2])

procedure section
    update EXEMPLAIRE
    set DATE_EMPRUNT = null
        MAT_EMPRUNTEUR = null,
        CODE_PROJET = null,
    where NUMERO_OUVRAGE = :NOU and NUMERO_D_ORDRE = :NOR;
    if SQLCODE <> 0 then  ...

end-module DELETE_EMPRUNT



module   CREATE_EMPRUNT   (NOU:char[10];   NOR:char[2];   MAT:char[10];
PRO:char[3])
declare section
    TODAY : date;
procedure section
    get_date(TODAY)
    update EXEMPLAIRE
    set DATE_EMPRUNT = :TODAY,
        MAT_EMPRUNTEUR = :MAT,
        CODE_PROJET = :PRO,
    where NUMERO_OUVRAGE = :NOU and NUMERO_D_ORDRE = :NOR;
    if SQLCODE <> 0 then  ...

end-module CREATE_EMPRUNT

...
end-program
```

15.3 REVERSE ENGINEERING STRATEGY

Let us recall the main processes in database reverse engineering when a global DMS-DDL schema is available.

The two main phases are *Data Structure Extraction*, through which the complete DMS-compliant data structures, together with their hidden integrity constraints, are elicited, and *Data Structure Conceptualization*, that builds a possible conceptual schema for these structures.

The **Data Structure Extraction** process will be made of two lower-level processes, namely *Global Schema Extraction*, that parses the DMS-DDL source code to extract declared data structures, and *Schema Refinement*, that analyses the procedural source code in order to refine these data structures. The end result is the DMS-compliant optimized logical schema.

The **Data Structure Conceptualization** process is decomposed into *Basic Conceptualization*, that translates and eliminates DMS-oriented and performance-oriented constructs, and that results into a first conceptual schema, and *Conceptual Normalization*, that refines the schema in order to give it clarity, minimality and self-expressiveness characteristics.

The final result is a possible conceptual schema for the source physical database description.

15.4 DATA STRUCTURE EXTRACTION

The objective of this phase is to make the RDBMS data structures and constraints explicit.

15.4.1 Global schema extraction

The SQL code is parsed and translated into data structures expressed in the ER model.

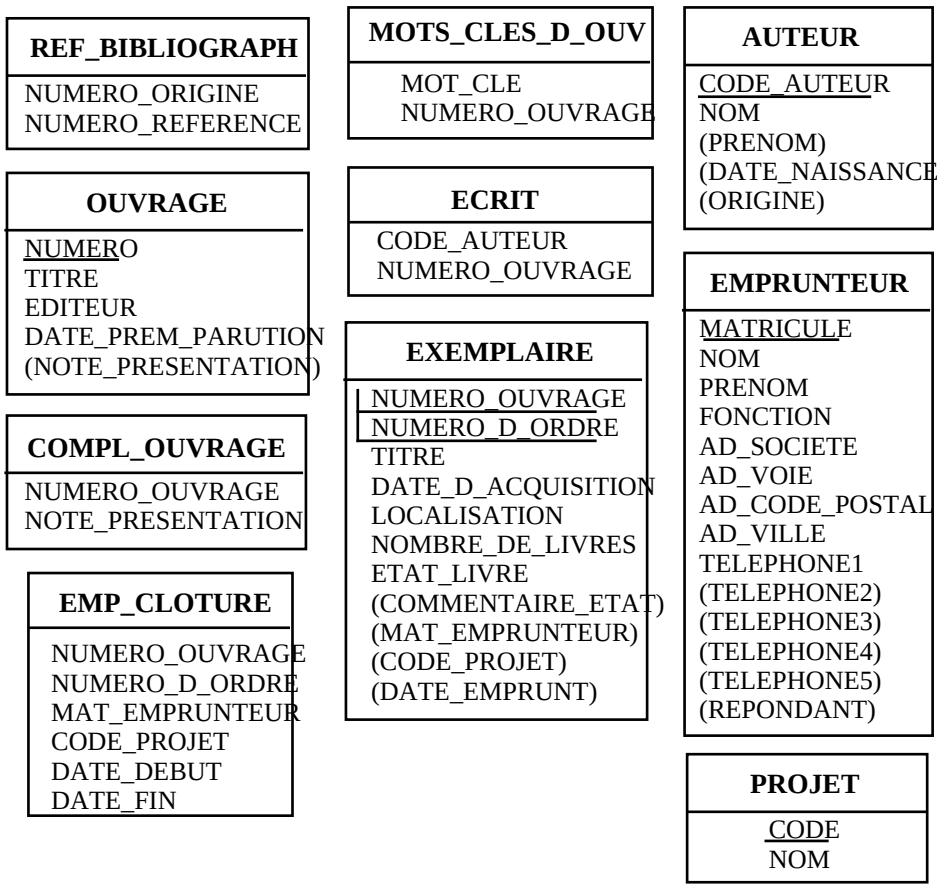
15.4.1.1 SQL-DDL logical code analysis

Analysis of the logical SQL-DDL source code

Each table is interpreted as a (physical) entity type, the attributes of which being the columns of the table. The `not null` clause translates into a mandatory property of the corresponding attribute, while its absence specifies optional attributes.

The `primary keys` define identifiers of these entity types, while `foreign keys` define reference attributes with their referential constraints.

The result of parsing the SQL-DDL text appears as follows.



REFERENTIAL CONSTRAINTS

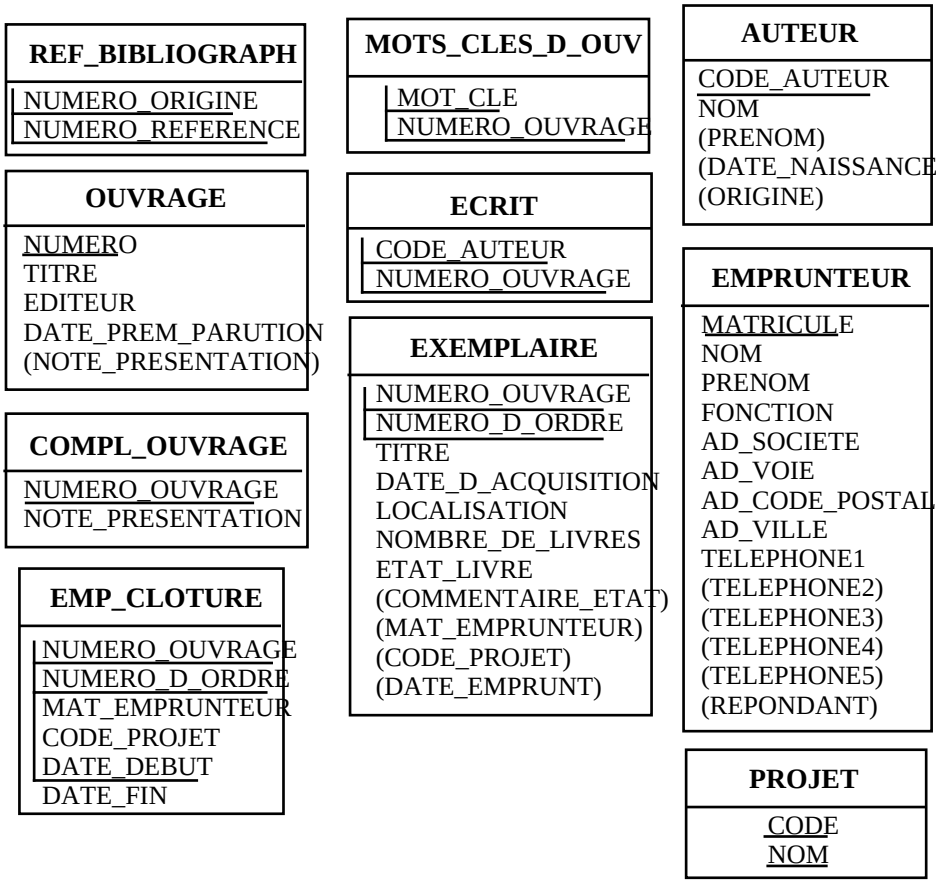
REF_BIBLIOGRAPH.NUMERO_ORIGINE **in** OUVRAGE.NUMERO
REF_BIBLIOGRAPH.NUMERO_REFERENCE **in** OUVRAGE.NUMERO
MOTS_CLES_D_OUV.NUMERO_OUVRAGE **in** OUVRAGE.NUMERO
ECRIT.NUMERO_OUVRAGE **in** OUVRAGE.NUMERO
ECRIT.CODE_AUTEUR **in** AUTEUR.CODE_AUTEUR
EXEMPLAIRE.NUMERO_OUVRAGE **in** OUVRAGE.NUMERO
EXEMPLAIRE.MAT_EMPRUNTEUR **in** EMPRUNTEUR.MATRICULE
EXEMPLAIRE.CODE_PROJET **in** PROJET.CODE
EMP_CLOTURE.(NUMERO_OUVRAGE, NUMERO_D_ORDRE)
in EXEMPLAIRE.(NUMERO_OUVRAGE, NUMERO_D_ORDRE)
EMP_CLOTURE.MAT_EMPRUNTEUR **in** EMPRUNTEUR.MATRICULE
EMP_CLOTURE.CODE_PROJET **in** PROJET.CODE
EMPRUNTEUR.REPONDANT **in** EMPRUNTEUR.MATRICULE
COMPL_OUVRAGE.NUMERO_OUVRAGE **in** OUVRAGE.NUMERO

15.4.1.2 SQL-DDL physical code analysis

Analysis of the physical SQL-DDL source code

The analysis of the physical schema (namely through the `create unique index` statements) leads to additional identifiers for the entity types. The following schema

expresses all the data structures that are defined in the SQL-DDL source code (the referential constraints are as described hereabove).



15.4.2 Schema refinement

The procedural source code is searched for evidences of additional structures and integrity constraints.

15.4.2.1 Procedural code analysis (program CHECK_BIBLIO)

This program appears as a validation procedure that checks the AUTEUR table to detect the rows that have no matching rows in the ECRIT table. The identifier values of these AUTEUR rows are recorded in the ERROR_FILE file. This behaviour and these names suggest that these AUTEUR rows are to be considered as incorrect. Therefore, we can state that any AUTEUR row must have at least one matching row in ECRIT[2] :

[2] Note that this constraint is not a referential constraint since the referenced attribute do not make a (primary) identifier.

```
AUTEUR.CODE_AUTEUR  in  ECRIT.CODE_AUTEUR
```

15.4.2.2 Procedural code analysis (program BIBLIO-MANAGEMENT)

This program collects the modules that manage the contents of table EXEMPLAIRE.

Modules CREATE_EXEMPLAIRE and UPDATE_EXEMPLAIRE

These modules are clearly in charge of controlling the insertion and updating of EXEMPLAIRE rows. In particular, it appears that (1) the value of TITRE comes from the corresponding OUVRAGE raw, and not from the user; (2) when updating an EXEMPLAIRE row, the value of TITRE cannot be changed by the user. In short, this value is a copy of the value of TITRE of the corresponding OUVRAGE raw. To be sure, the procedural code dedicated to the management of table OUVRAGE should be analysed to be sure that these modules are the only legal way to update these tables. Hence the following redundancy constraint :

```
e.TITRE = OUVRAGE(NUMERO = e.NUMERO_OUVRAGE).TITRE, ∀e  EXEMPLAIRE
```

Modules CREATE_EMPRUNT and UPDATE_EMPRUNT

These modules are dedicated to the management of three attributes of EXEMPLAIRE, namely MAT_EMPRUNTEUR, CODE_PROJET and DATE_EMPRUNT. We observe that,

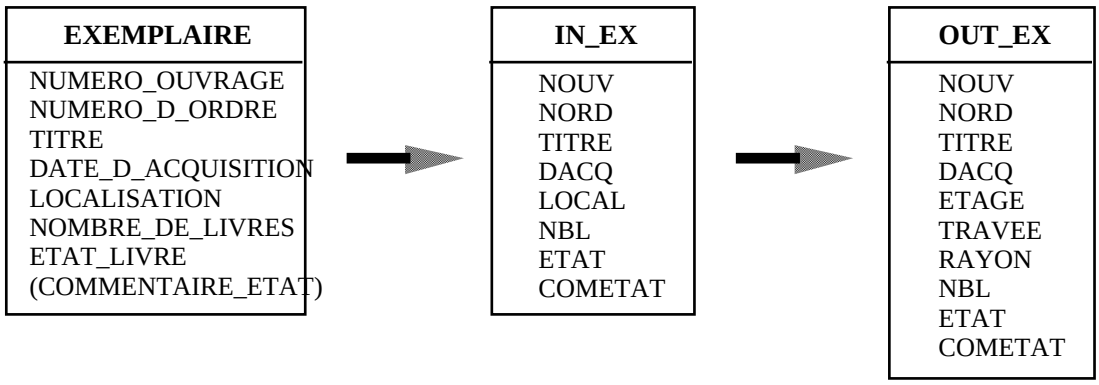
- CREATE_EMPRUNT sets these three attributes to external values,
- DELETE_EMPRUNT sets these three attributes to null,
- there is no other way to update these attributes; for instance, the management modules analyzed hereabove ignore these attributes (this should be confirmed by a more in-depth analysis of the procedural code and of the database contents).

We can conclude that these attributes, either have significant values, or have no values. This property is expressed as a coexistence group as follows :

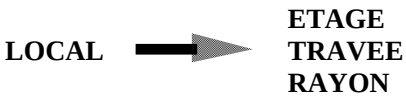
```
coexist(EXEMPLAIRE): MAT_EMPRUNTEUR,CODE_PROJET,DATE_EMPRUNT
```

Module DISPLAY_EXEMPLAIRE

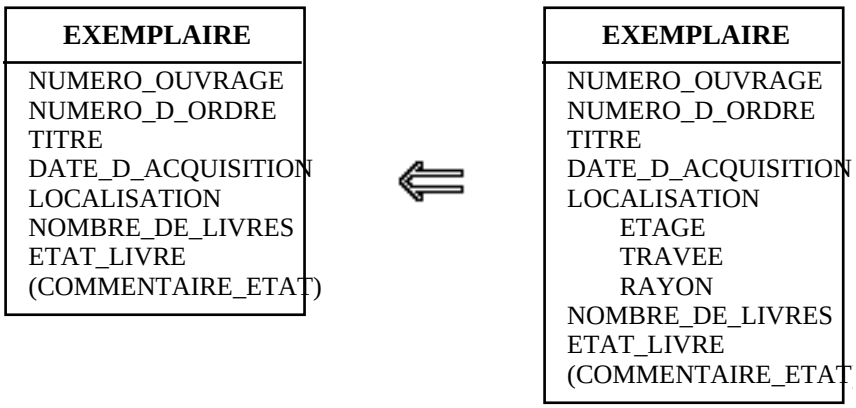
This module display the contents of an identified EXEMPLAIRE row in a readable form. It makes use of two internal buffers, namely IN_EX, into which a row is read, and OUT_EX, from which the row is displayed. The data flows can be sketched as follows :



Though they are intended to contain the same data, these buffers have different structures.. Indeed, input field LOCAL is replaced by output fields ETAGE, TRAVEE, RAYON.. Computing the physical positions and lengths of these fields shows that the output fields cover exactly the input field. Therefore, the output fields, which represent the user's view, can be considered as a refined structure for the input field.

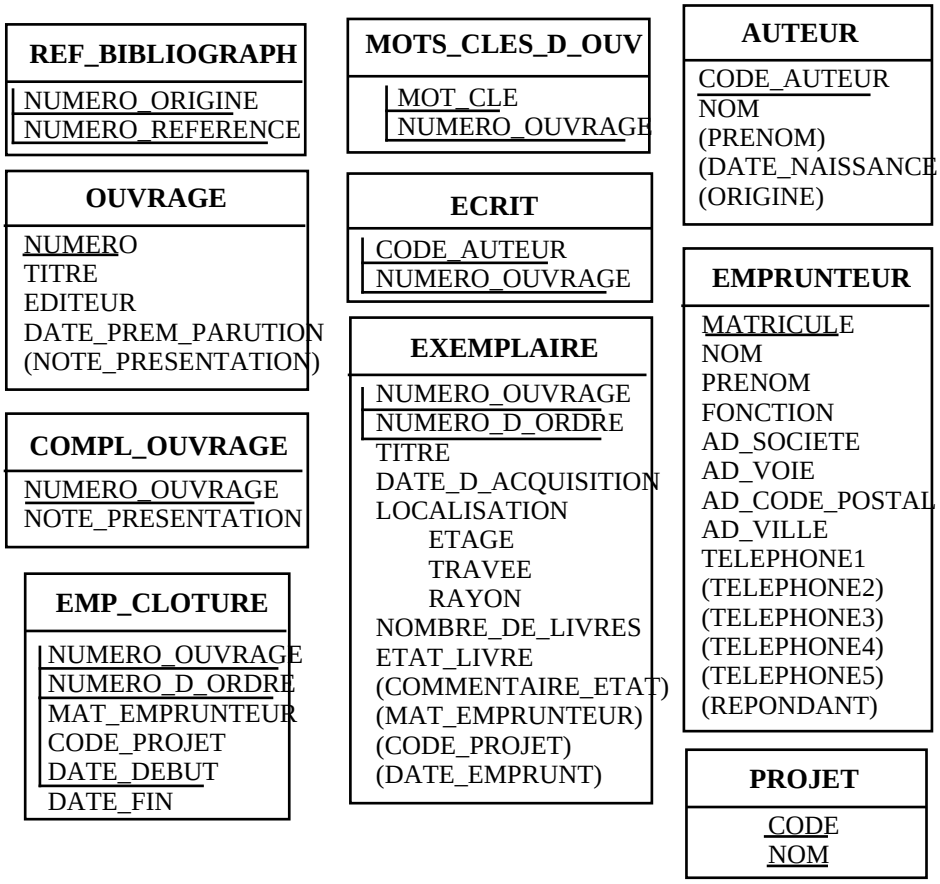


We can infer that the input column, LOCALISATION, is perceived by users as being decomposed into ETAGE, TRAVEE, RAYON.



Let us observe that a similar analysis of modules CREATE_EXEMPLAIRE and UPDATE_ EXEMPLAIRE would have lead to the same conclusion.

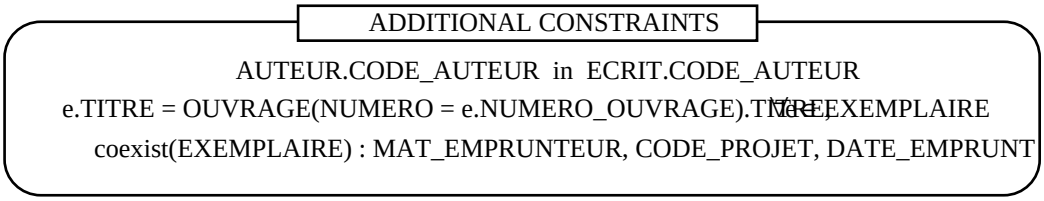
At this stage, the source code analysis process can be considered finished. It results in the following **Optimized SQL-compliant logical schema**[3].



REFERENTIAL CONSTRAINTS

REF_BIBLIOGRAPH.NUMERO_ORIGINE **in** OUVRAGE.NUMERO
REF_BIBLIOGRAPH.NUMERO_REFERENCE **in** OUVRAGE.NUMERO
MOTS_CLES_D_OUV.NUMERO_OUVRAGE **in** OUVRAGE.NUMERO
ECRIT.NUMERO_OUVRAGE **in** OUVRAGE.NUMERO
ECRIT.CODE_AUTEUR **in** AUTEUR.CODE_AUTEUR
EXEMPLAIRE.NUMERO_OUVRAGE **in** OUVRAGE.NUMERO
EXEMPLAIRE.MAT_EMPRUNTEUR **in** EMPRUNTEUR.MATRICULE
EXEMPLAIRE.CODE_PROJET **in** PROJET.CODE
EMP_CLOTURE.(NUMERO_OUVRAGE, NUMERO_D_ORDRE)
in EXEMPLAIRE.(NUMERO_OUVRAGE, NUMERO_D_ORDRE)
EMP_CLOTURE.MAT_EMPRUNTEUR **in** EMPRUNTEUR.MATRICULE
EMP_CLOTURE.CODE_PROJET **in** PROJET.CODE
EMPRUNTEUR.REPONDANT **in** EMPRUNTEUR.MATRICULE
COMPL_OUVRAGE.NUMERO_OUVRAGE **in** OUVRAGE.NUMERO

[3] It could be objected that compound attribute LOCALISATION is not RDBMS-compliant and that its decomposition should be specified as an additional constraint instead. The argument for the expression adopted in this schema is mainly simplicity.



15.5 DATA STRUCTURE CONCEPTUALIZATION

The objective of this second phase is to make the semantics of the DMS-compliant schema explicit. It consists in two major processes, namely *Basic conceptualization* and *Conceptual normalization*.

15.5.1 Basic conceptualization

Through this process, the relational structures are expressed in a DMS-independent schema through the *Untranslation* process, and the optimization constructs are detected and removed through the *De-optimization* process.

15.5.1.1 SQL-untranslation

We shall concentrate on reference attributes and serial attributes processing.

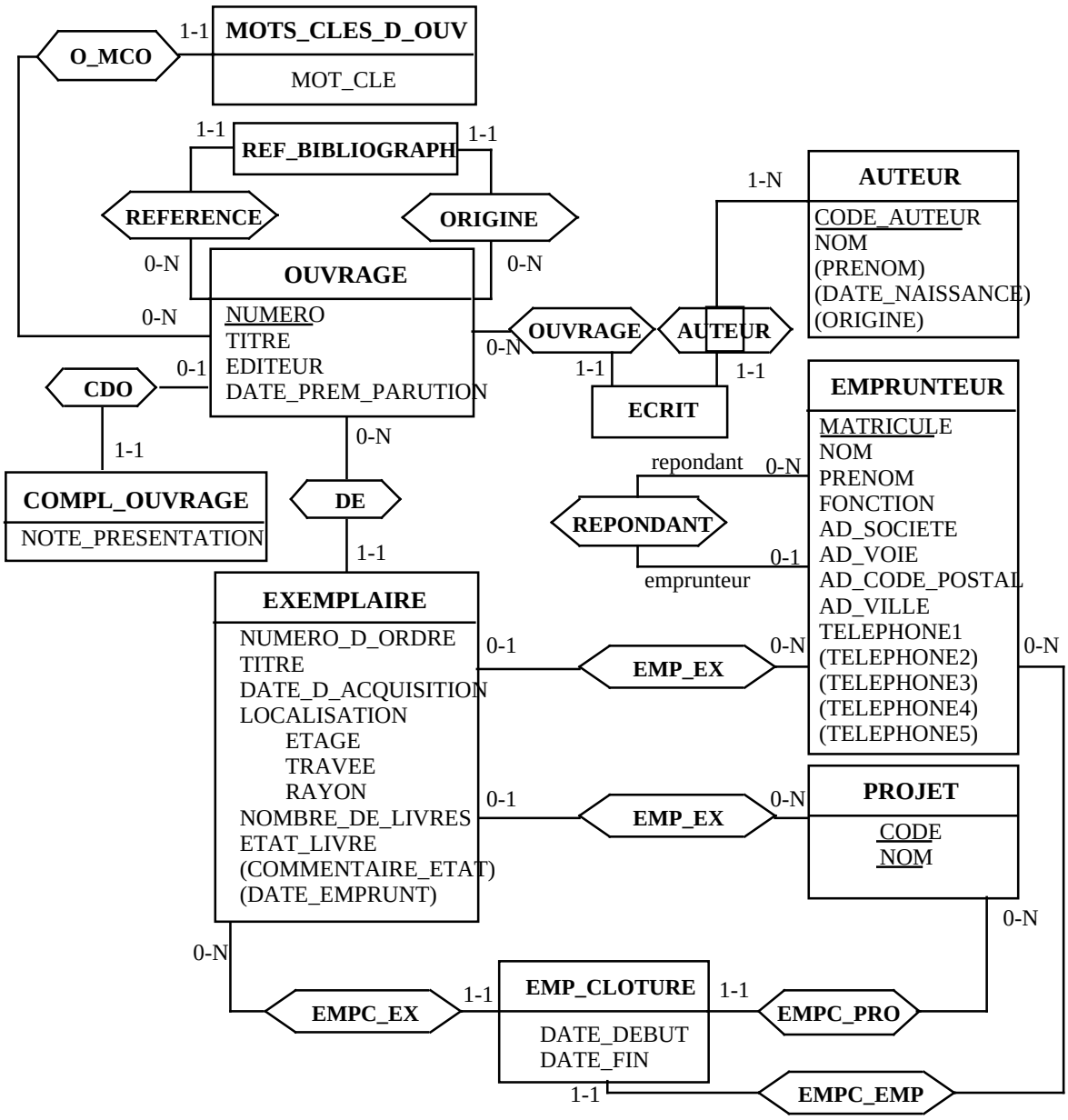
REFERENCE ATTRIBUTES

Each group of reference attributes associated with a referential constraint is transformed into a functional (one-to-many or one-to-one) rel-type.

When such a group is an identifier for its entity type, then the role taken by the referenced entity type in the new rel-type has cardinality 0-1 or 1-1 (e.g. EMP_EX).

If the group is optional then the cardinality of the source entity type is 0-1 (e.g. REPONDANT).

A significant name has to be defined for each new rel-type.



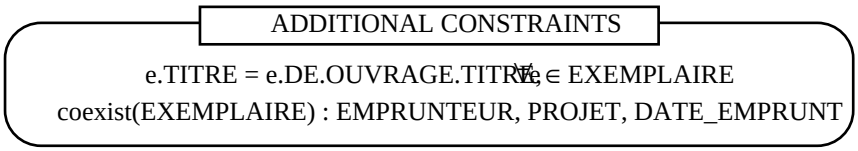
When a reference group is a component of an identifier, it is replaced in the latter by the role through which the referenced entity type participates in the new rel-type.

IDENTIFIERS

id(EXEMPLAIRE): OUVRAGE, NUMERO_D_ORDRE
id(ECRIT) : OUVRAGE, AUTEUR
id(REF_BIBLIOGRAPH): REFERENCE, ORIGINE
id(EMP_CLOTURE) : EXEMPLAIRE, DATE_DEBUT
id(MOTS_CLES_D_OUV) : OUVRAGE, MOT_CLE

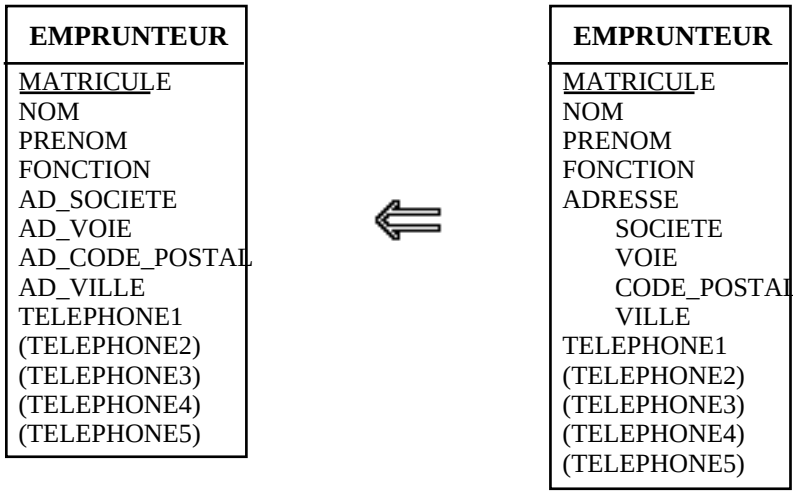
The additional constraints are converted as well. The first one, stating that each AUTEUR entity must match at least one ECRIT entity on their CODE_AUTEUR values is translated

into cardinality 1-N of AUTEUR. The second and third constraints are translated as follows.

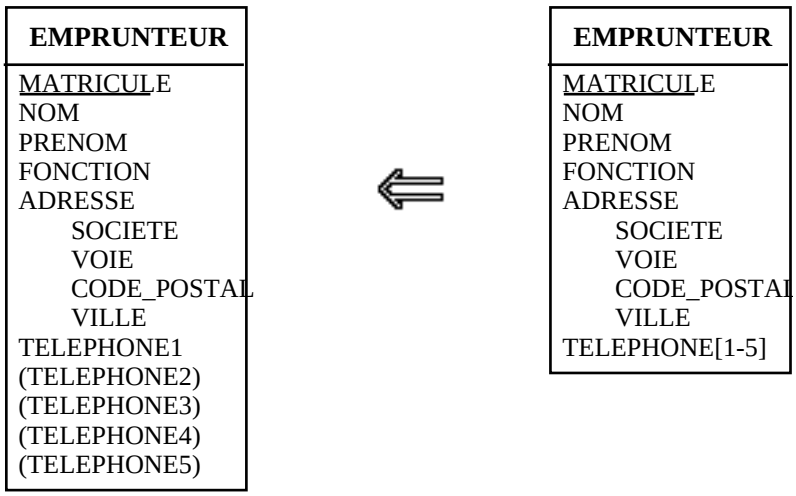


SERIAL ATTRIBUTES

Entity type EMPRUNTEUR exhibits a sequence of attributes that share a common prefix, AD_. This pattern evokes a logical relationship that can be made explicit by grouping them as compound attribute ADRESSE. This name derives from the prefix, but must be confirmed by additional investigation.



Another sequence of attribute names can be found in this entity type : TELEPHONE1 to TELEPHONE5. Here, the prefix appears as a complete name, while the variable part is merely a sequence number that provides name uniqueness. This pattern evokes a multivalued attribute whose name is the common prefix, namely TELEPHONE. Its cardinality is easy to determine : one source attribute is mandatory while the others are optional, hence cardinality [1-5].



15.5.1.2 De-optimization

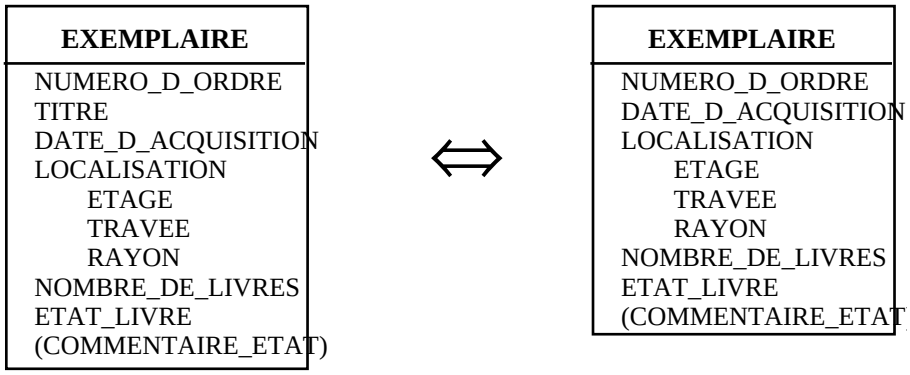
The resulting schema is then analysed in order to detect possible traces of optimization constructs. Three families of techniques can be used.

NORMALIZATION

No unnormalized structures can be detected on the basis of the available information.

STRUCTURAL REDUNDANCY

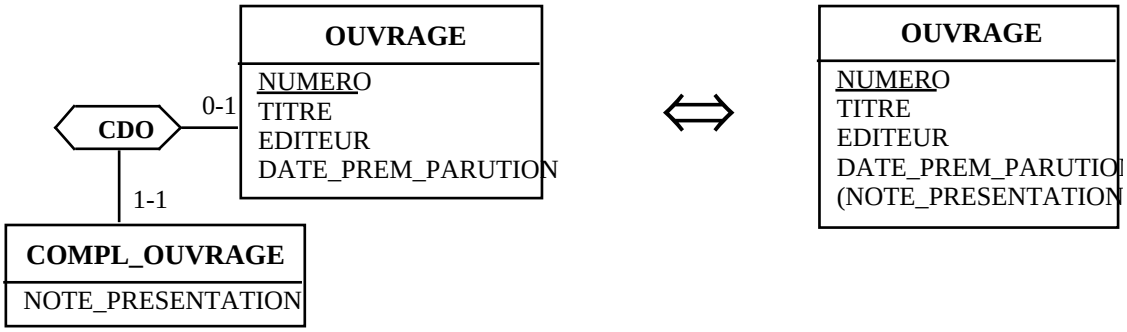
The additional constraints expresses that TITRE in each EXEMPLAIRE entity has the same value as that of TITRE in the corresponding OUVRAGE entity. This means that EXEMPLAIRE.TITRE is redundant with OUVRAGE.TITRE (and not conversely). The former can be discarded as a derived attribute.



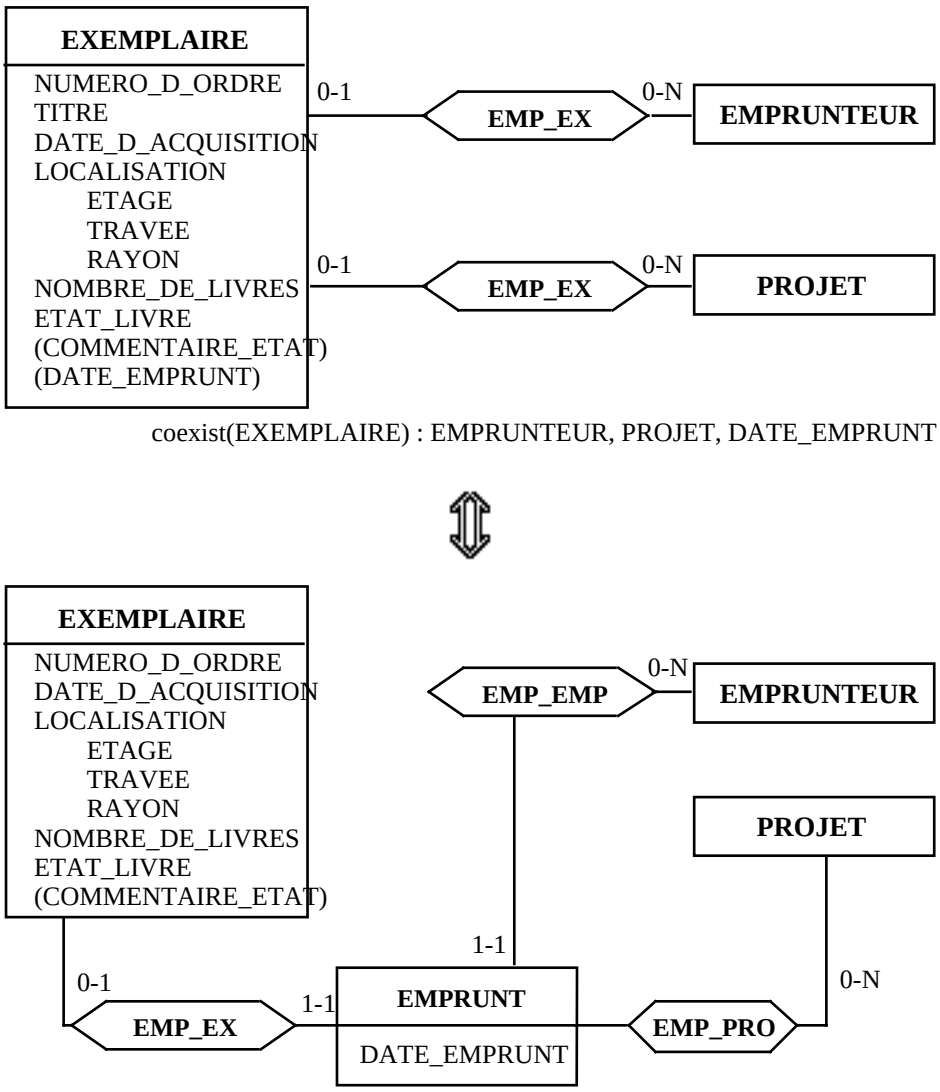
$\forall e \in$ EXEMPLAIRE e.TITRE = e.DE.OUVRAGE.TITRE

RESTRUCTURATION

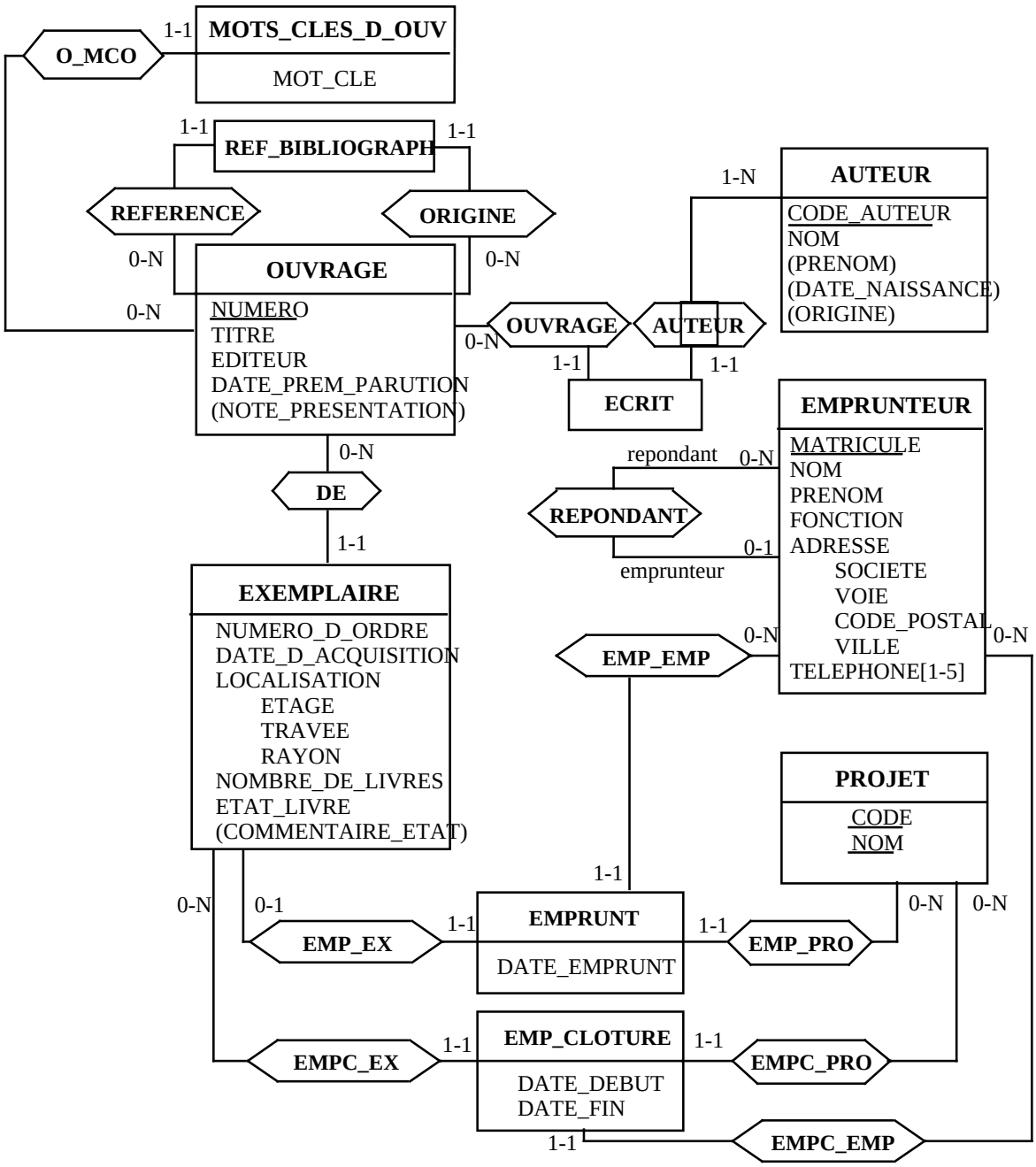
The presence of a one-to-one rel-type between two entity types, one of which having no identifier, is an evidence of the result of a possible *vertical partitioning* restructuration. This pattern obviously does not concern EMP_EX : merging EXEMPLAIRE and EMPRUNT would lead to a complex coexistence constraint and to less clarity. On the contrary, COMPL_OUVRAGE appears as a large, optional, fragment of OUVRAGE. Both can be merged.



Traces of an optimization *vertical merging* can be found through the coexistence group that is associated with EXEMPLAIRE. The components of the group can be extracted through *vertical partitioning* to form a new entity type. A name for the latter can be suggested by the name of the modules that manage these components, namely EMPRUNT.



The resulting schema is a first conceptual schema, i.e. a schema that is DMS-independent and from which all the (detected) performance-oriented constructs have been removed.



IDENTIFIERS

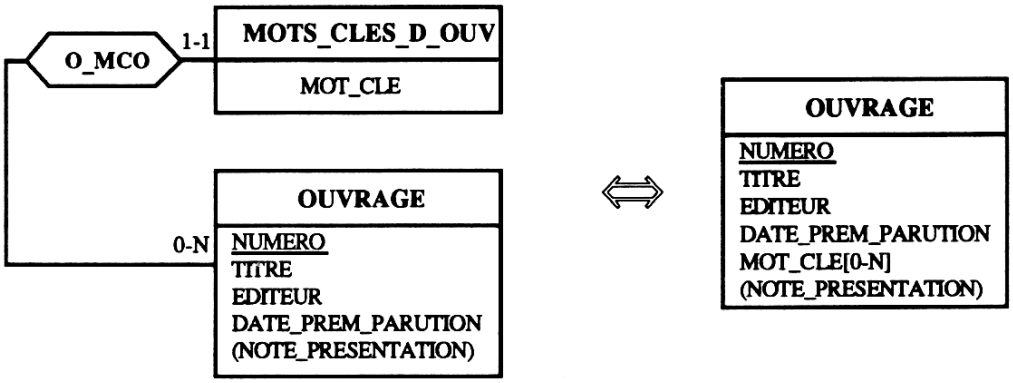
id(EXEMPLAIRE): OUVRAGE, NUMERO_D_ORDRE
id(ECRIT) : OUVRAGE, AUTEUR
id(REF_BIBLIOGRAPH): REFERENCE, ORIGINE
id(EMP_CLOTURE) : EXEMPLAIRE, DATE_DEBUT
id(MOTS_CLES_D_OUV) : OUVRAGE, MOT_CLE

15.5.2 Conceptual normalization

The first conceptual schema can be restructured in order to give it a higher degree of clarity, minimality and self-expressiveness[4] [BATINI,92]. The schema will be search for three common constructs that can be reduced for greater simplicity and clarity, namely *attribute entity types*, *IS-A one-to-one rel-types* and *relationship entity types*.

ATTRIBUTE ENTITY TYPES

Entity type MOT_CLE_D_OUV have one attribute only, and appears as dependent on OUVRAGE. It can be considered as the instance representation of multivalued attribute MOT_CLE of OUVRAGE.



IS-A ONE-TO-ONE REL-TYPES

EMP_EX between EXEMPLAIRE and EMPRUNT is a one-to-one rel-type whose role can be questioned : does it represent an IS-A relation between generic entity type EXEMPLAIRE and specific entity type EMPRUNT ? Is EMPRUNT a category of EXEMPLAIRE ? By considering the semantics of EMPRUNT, it appears that it describes the last occurrence of an operation on an EXEMPLAIRE rather than a specific state of EXEMPLAIRE. It is then decided not to replace this rel-type.

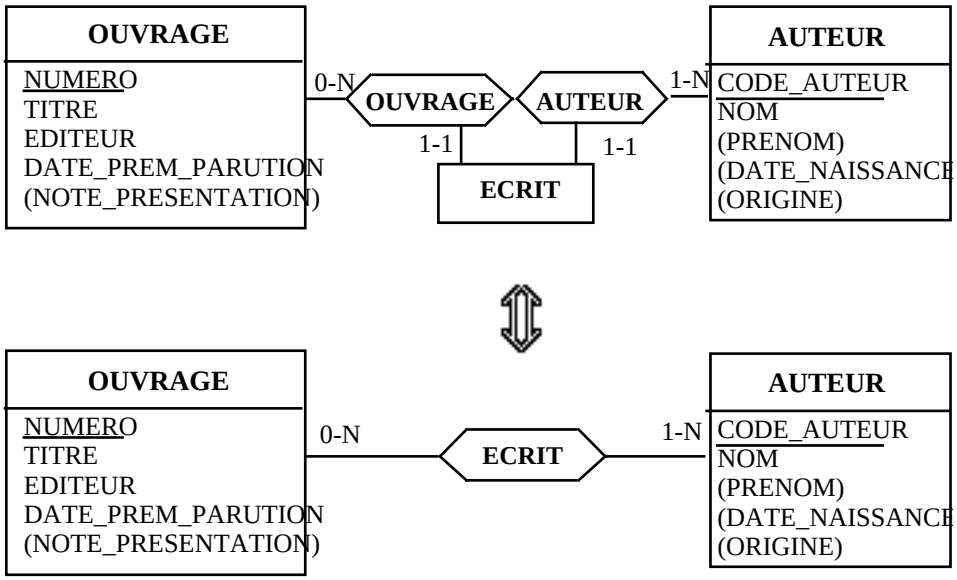
RELATIONSHIP ENTITY TYPES

An entity type can be defined as a link between other entity types. Generally, it has no or few attributes, it participates in at least two roles of functional rel-types, and its cardinalities are 1-1.

---

4 See Batini, Ceri, Navathe, *Conceptual Database Design*, Benjamin-Cummings, 1992

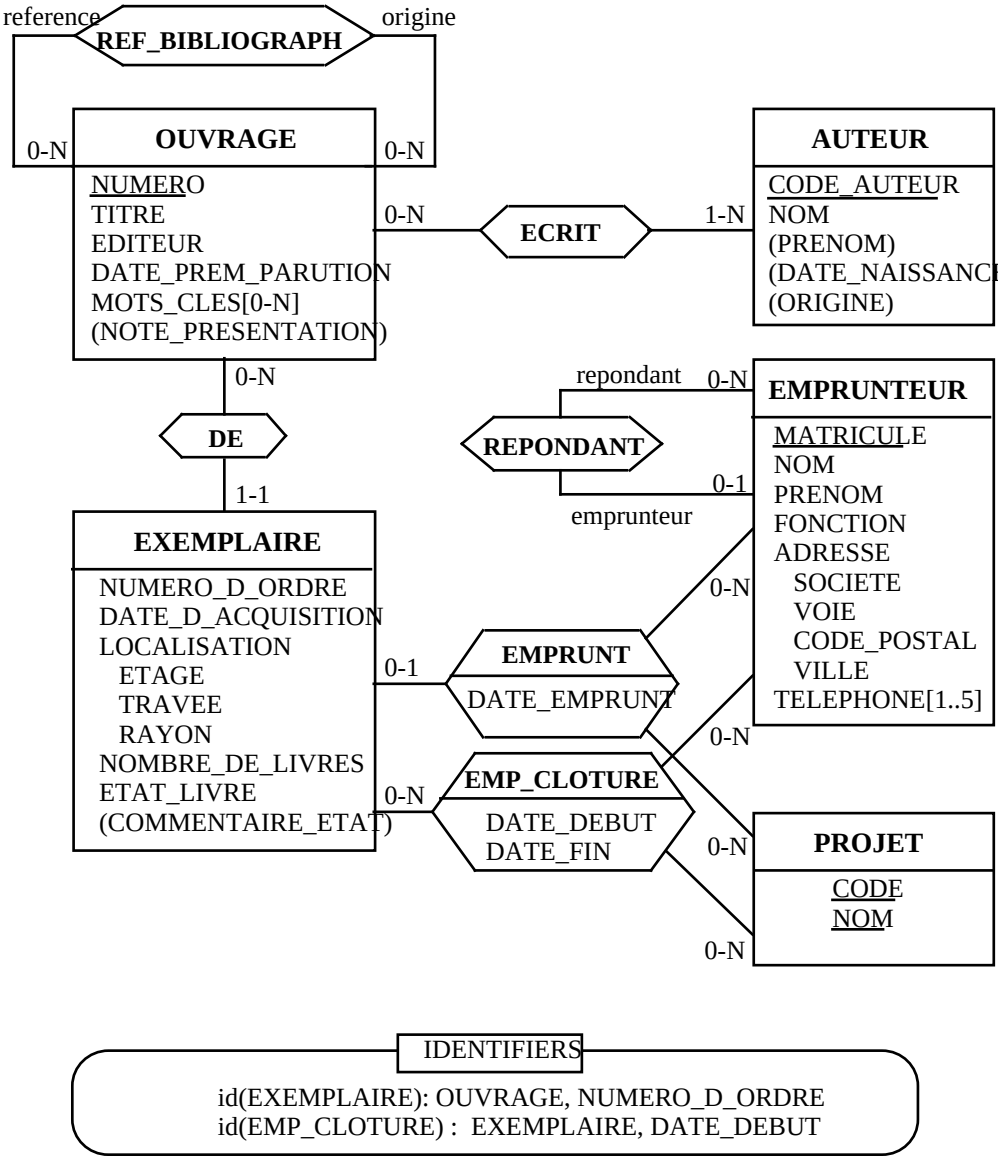
If desired, such an entity type can be replaced by a rel-type. ECRIT and REF_BIBLIOGRAPH are clear candidates for such a transformation. The replacement is shown for rel-type ECRIT.



Rel-types EMPRUNT and EMPRUNT_CLOTURE satisfy this pattern as well. They can be transformed into rel-types EMPRUNT and EMPRUNT_CLOTURE if the analyst finds this expression more natural. However, the latter transformations may not gain a general agreement, in which case the current entity-type format will be kept without loss of semantics.

15.6 THE FINAL CONCEPTUAL SCHEMA

The conceptual schema obtained so far should be validated against user's perception. It could appear as follows.



Chapter 16

A SHORT COBOL CASE STUDY

This second case study processes a set of COBOL file declarations, from which a conceptual schema will be recovered.

16.1 PRESENTATION

We are provided with the description of three standard files, expressed into COBOL program fragments. To simplify the presentation, we will ignore the procedural parts of the programs themselves, except when needed, in which case we will simply make comments on how additional information can be found. Anyway, extracting properties from the source code of the programs has been illustrated in chapter 15 already. These reasonings can be considered as general, and applicable in this context.

16.2 THE SOURCE CODE TEXTS

16.2.2 File and record description

```
ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.
        SELECT PRODUCTS ASSIGN TO DSK:FSP ;
           ORGANIZATION IS INDEXED;
           RECORD KEY IS P-pron;
           ALTERNATE RECORD KEY IS P-store-ref WITH DUPLICATES.

        SELECT MATERIALS ASSIGN TO DSK:FSM ;
           ORGANIZATION IS INDEXED;
           RECORD KEY IS M-matn.

        SELECT PRODUCTION ASSIGN TO DSK:FUP ;
           ORGANIZATION IS INDEXED;
           RECORD KEY IS U-id.


DATA DIVISION.

FILE SECTION.

FD   PRODUCTS;  LABEL RECORDS ARE STANDARD.
        RECORD IS PRODUCT.

        01  PRODUCT.
           03 P-pron PIC IS 9(6).
           03 P-name PIC IS X(15).
           03 P-packaging PIC IS X(15).
           03 P-store-ref PIC IS 9(2).
           03 P-store-local PIC IS X(15).
           03 P-availq PIC IS 9(4).


FD   MATERIALS;  LABEL RECORDS ARE STANDARD.
        RECORD IS MATERIAL.

        01  MATERIAL.
           03 M-matn PIC IS 9(5).
           03 M-mat-type PIC IS X;
              88 M-raw-mat; value is 'R';
              88 M-cons-mat; value is 'C'.
```

```
           03 M-name PIC IS X(15).
           03 M-cat PIC IS X(15).
           03 M-availq PIC IS 9(4).
           03 M-ord-num PIC IS 9(2).
           03 M-order OCCURS 1 TO 10; DEPENDING ON M-ord-num.
              05 M-ordn PIC IS 9(8).
              05 M-date PIC IS X(6).
              05 M-ordq PIC IS 9(4).
              05 M-supplq PIC IS 9(4).
              05 M-supplier-ref PIC IS 9(4).
           03 M-total-cons PIC IS 9(2).
           03 M-stock-num PIC IS 9(2).
           03 M-id-stock  OCCURS 1 TO 20; DEPENDING ON M-stock-num;
                          PIC IS 9(2).

FD  PRODUCTION;  LABEL RECORDS ARE STANDARD.
        RECORD IS PROD-UNIT.

        01 PROD-UNIT.
           03 U-id.
              05 U-pu-id PIC IS 9(3).
              05 FILLER PIC IS 9(5); VALUE IS 0.
           03 U-localisation PIC IS X(15).
           03 U-daily-prod-capac.
              05 U-mach-hours PIC IS 99.
              05 U-man-hours PIC IS 999.


        01 PU-INPUT REDEFINES PROD-UNIT.
           03 I-id.
              05 I-pu-id PIC IS 9(3).
              05 I-mat-ref PIC IS 9(5).
           03 I-standard-q PIC IS 9(4).
           03 FILLER PIC IS X(16).
```

16.2.2 The program

The procedural sections will be ignored. Indeed, their usage would be similar to what has been discussed in chapter 15. However, some reasonings will be based on information that could have been found in the programs or in file contents. They will be mentioned when needed.

16.3 REVERSE ENGINEERING STRATEGY

We will follow the methodology proposed in this manual. However, some flexibility will be suggested when conducting the standard processes. For instance, some activities of the two main processes will be carried out in alternately. However, we will present the example the standard way.

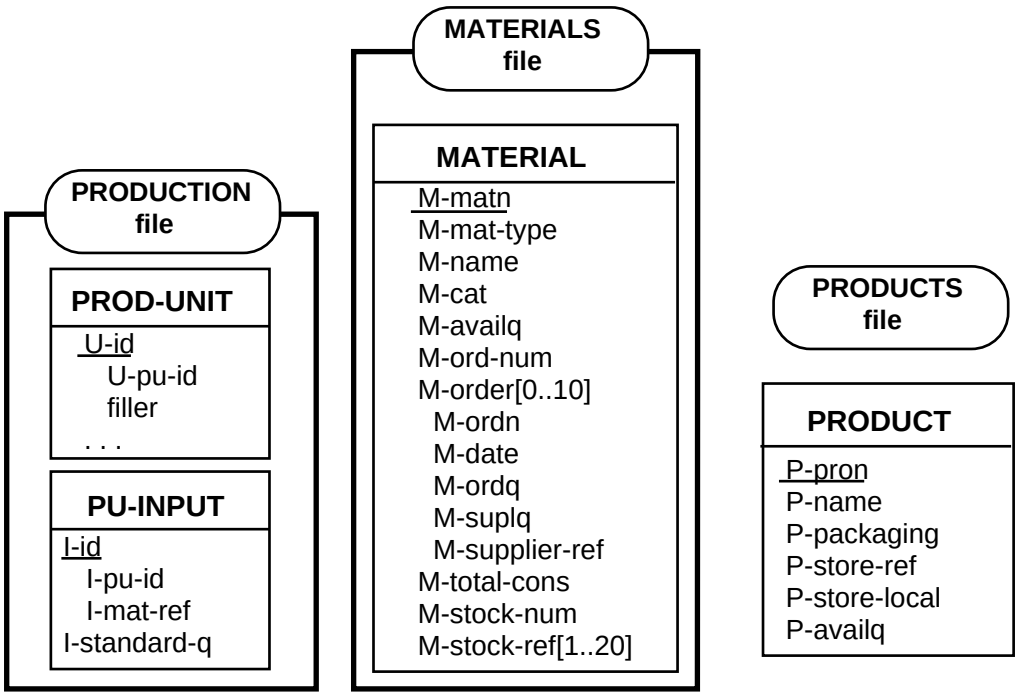
16.4 DATA STRUCTURE EXTRACTION

The objective of this phase is to make the COBOL data structures and constraints explicit.

16.4.1 Global schema extraction

The COBOL code is parsed and translated immediately into the following data structures. Some comments :

- The PRODUCTION file includes record type PROD-UNIT, itself redefined as PU-INPUT. This declaration has been understood as the specification of two record types.
- The *88 condition names* have been interpreted as the declaration of two major elements of the value domain of field M-mat-type of record type MATERIAL.
- The *depending on* clauses have not been considered. Indeed, they are redundant with the *occurs .. to ..* clause in detecting a varying multivalued attribute.
- Note the global identifier on the PRODUCTION file.




```
PROD-UNIT.U-id.filler = 0
MATERIAL.M-mat-type = {'R','C'}
id(PROD-UNIT + PU-INPUT): {U-id;I-id}
key(PRODUCT): P-store-ref
```

16.4.2 Schema refinement

The procedural source code, as well as the file contents, are searched for traces of additional structures and integrity constraints. Since we will ignore this source of information, we will mention from now on, whenever required, the additional information that could have been obtained by program source code analysis.

16.5 DATA STRUCTURE CONCEPTUALIZATION

The objective of this second phase is to make the semantics of this COBOL-compliant schema explicit. It consists in two major processes, namely *Basic conceptualization* and *Conceptual normalization*.

16.5.1 Basic conceptualization

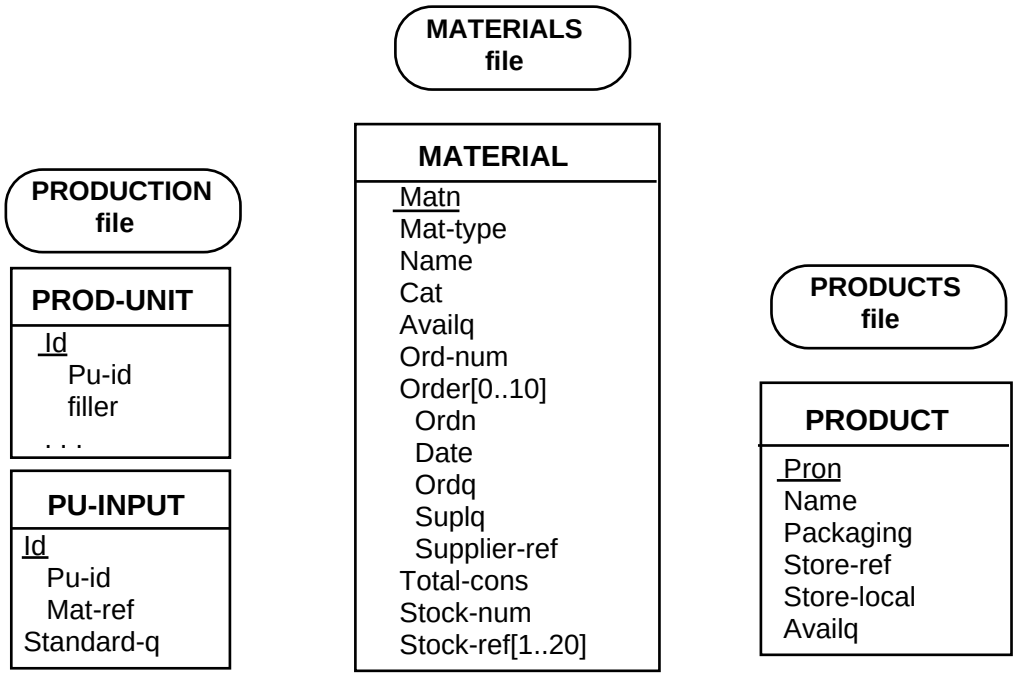
Through this process, the file structures are expressed in a COBOL-independent schema through the *Untranslation* process, and the optimization constructs are detected and

removed through the *De-optimization* process. Some preliminary processing of the field names can make the schema more readable.

16.5.1.1 Preliminary name processing

Clearly, the field names within a record type have been prefixed with a substring that is specific to this record type. This practice is common to make the field names unique in order to avoid qualified field identification, considered as too long by some programmers.

These prefix are removed without loss of information.



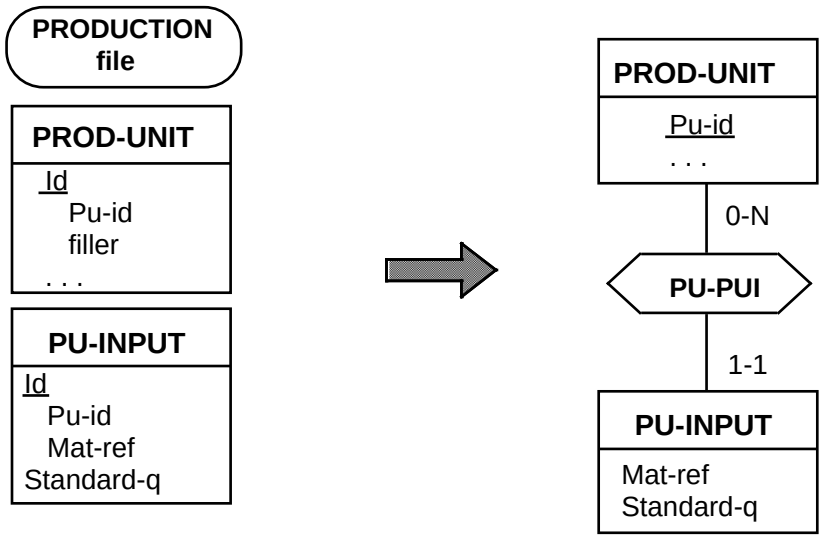

```
PROD-UNIT.Id.filler = 0
MATERIAL.Mat-type = {'R','C'}
id(PROD-UNIT + PU-INPUT): {Id;Id}
key(PRODUCT): Store-ref
```

16.5.1.2 COBOL-untranslation

We shall concentrate on reference attributes and serial attributes processing.

MULTI-RECORD-TYPE FILE

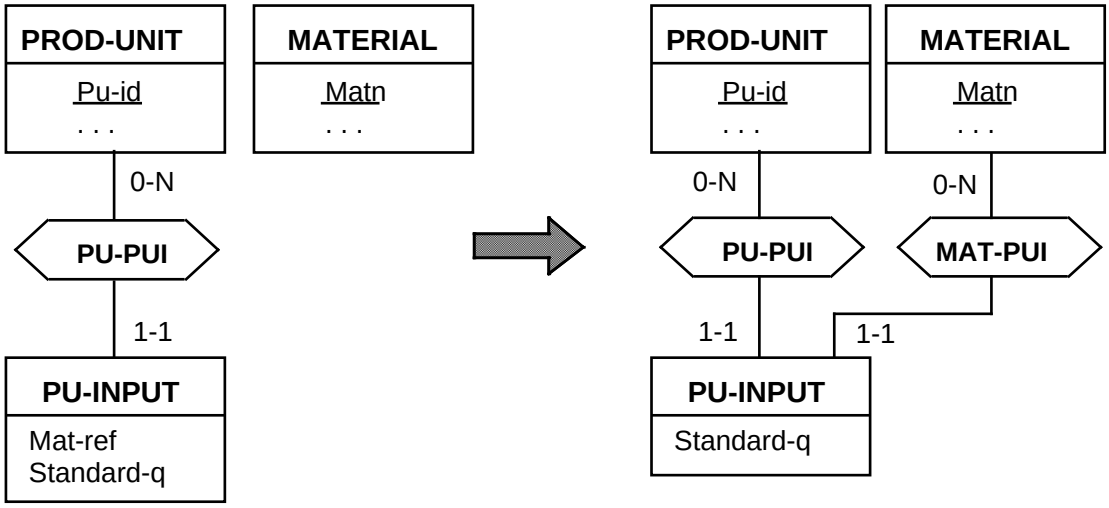
The PRODUCTION file includes a typical arrangement of records that strongly suggests the implementation of a one-to-many rel-type. This hypothesis can be checked through program analysis.




```
PROD-UNIT.Id.filler = 0                    id(PROD-INPUT): PROD-UNIT,Mat-ref
id(PROD-UNIT + PU-INPUT): {Id;Id}
```

REFERENCE ATTRIBUTE

Attribute name Mat-ref suggests the reference role of this attribute. Let us suppose that we have succeeded in proving that a reference constraint exists from Mat-ref to MATERIAL[1]. We can then transform this structure into a rel-type. The identifier of PU-UNIT has been modified accordingly.



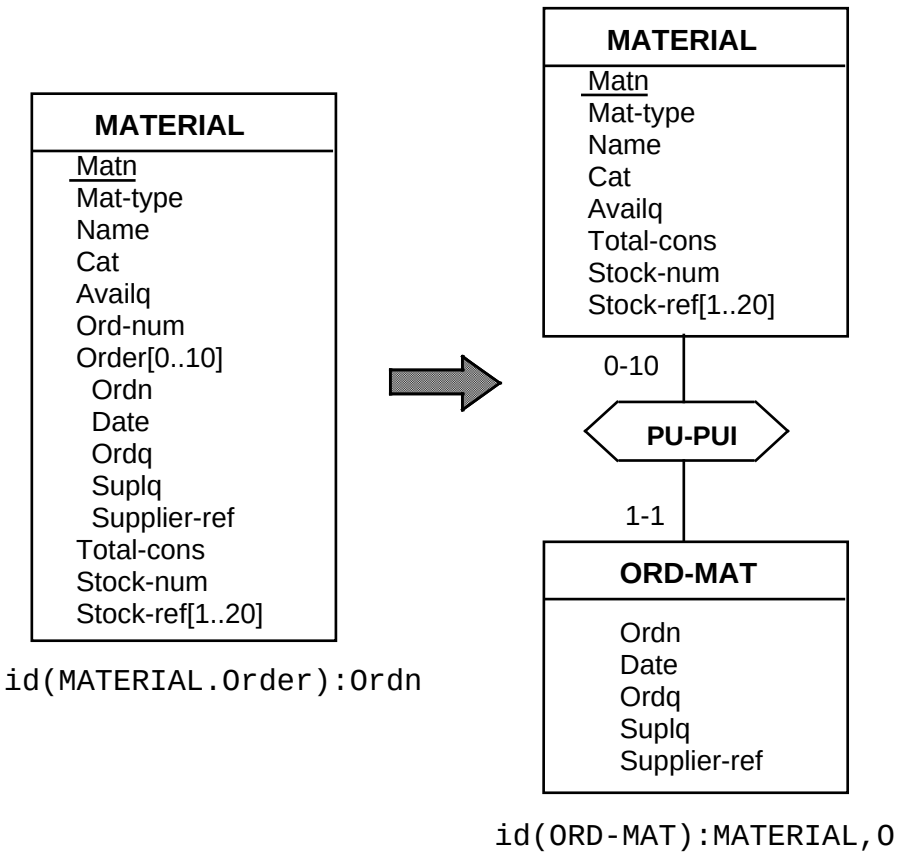

```
id(PU-INPUT): PROD-UNIT,Mat-ref        id(PU-INPUT): PROD-UNIT,MATERIAL
PU-INPUT.Mat-ref is-in MATERIAL.Matn
```

---

[1] This is an interesting example on the need for flexible strategies. Indeed, finding this referential constraint is an activity pertaining to the Data Structure Extraction process.

MULTIVALUED, COMPOUND ATRIBUTES

Entity type MATERIAL includes a complex attribute, Order, that can be examined in order to check its real nature. Let us suppose that, by eximination of the file contents and/or of the programs, we are convinced that all the Order.Ordn are distinct for each MATERIAL entity. We can then propose to express this multivalued attribute as a dependent entity type.



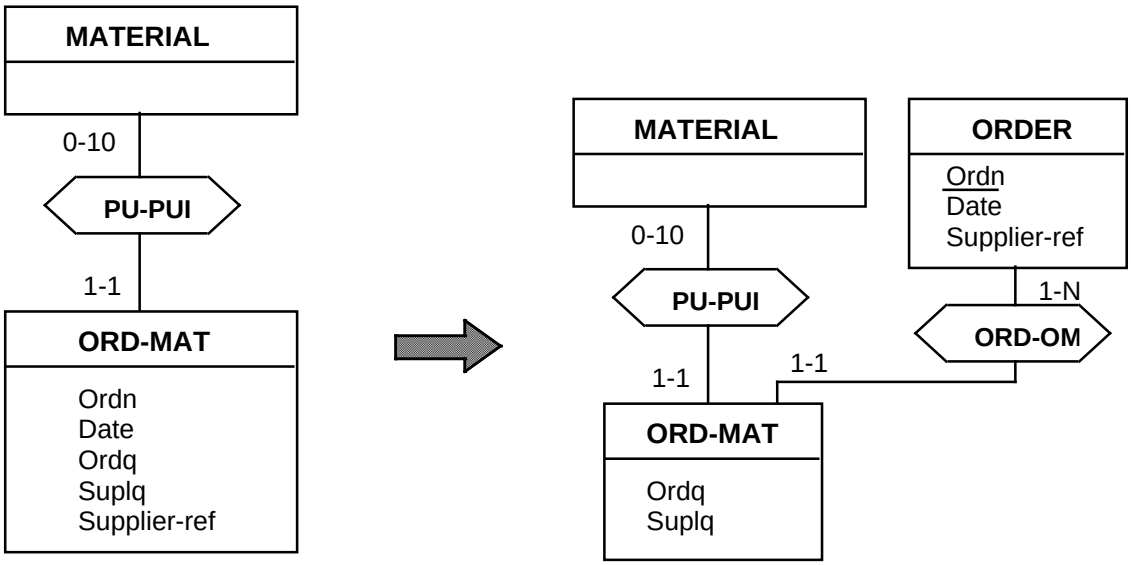
16.5.1.3 De-optimization

The resulting schema is then analysed in order to detect possible traces of optimization constructs.

NORMALIZATION

The ORD-MAT entity type includes an attribute, Ordn, whose name suggests some identifying property. Since it is not the identifier of its entity type, it is examined to check whether it identifies a subset of the other attributes. This can be done by analysing the

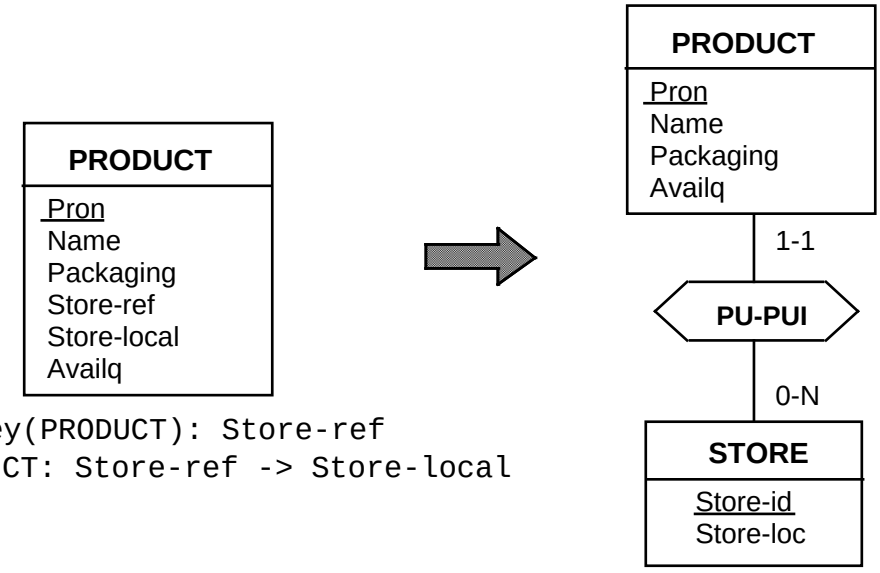
programs and the file contents. Let us suppose that such a property has been found. It is translated into a functional dependency. The schema can be normalized as follows.




```
id(ORD-MAT):MATERIAL,Ordn
ORD-MAT: Ordn -> Date,Supplier-ref
```

```
id(ORD-MAT):MATERIAL,ORDER
```

A similar pattern can be detected in entity type PRODUCT. Here, we are provided with an additional evidence : attribute Store-ref is an access key, which gives it a special status among the attributes of PRODUCT.



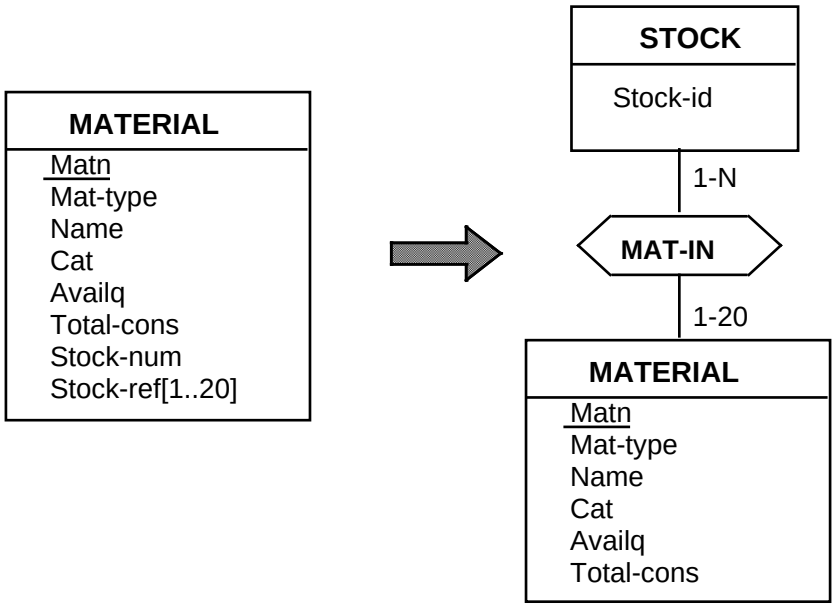

```
key(PRODUCT): Store-ref
PRODUCT: Store-ref -> Store-local
```

STRUCTURAL REDUNDANCY

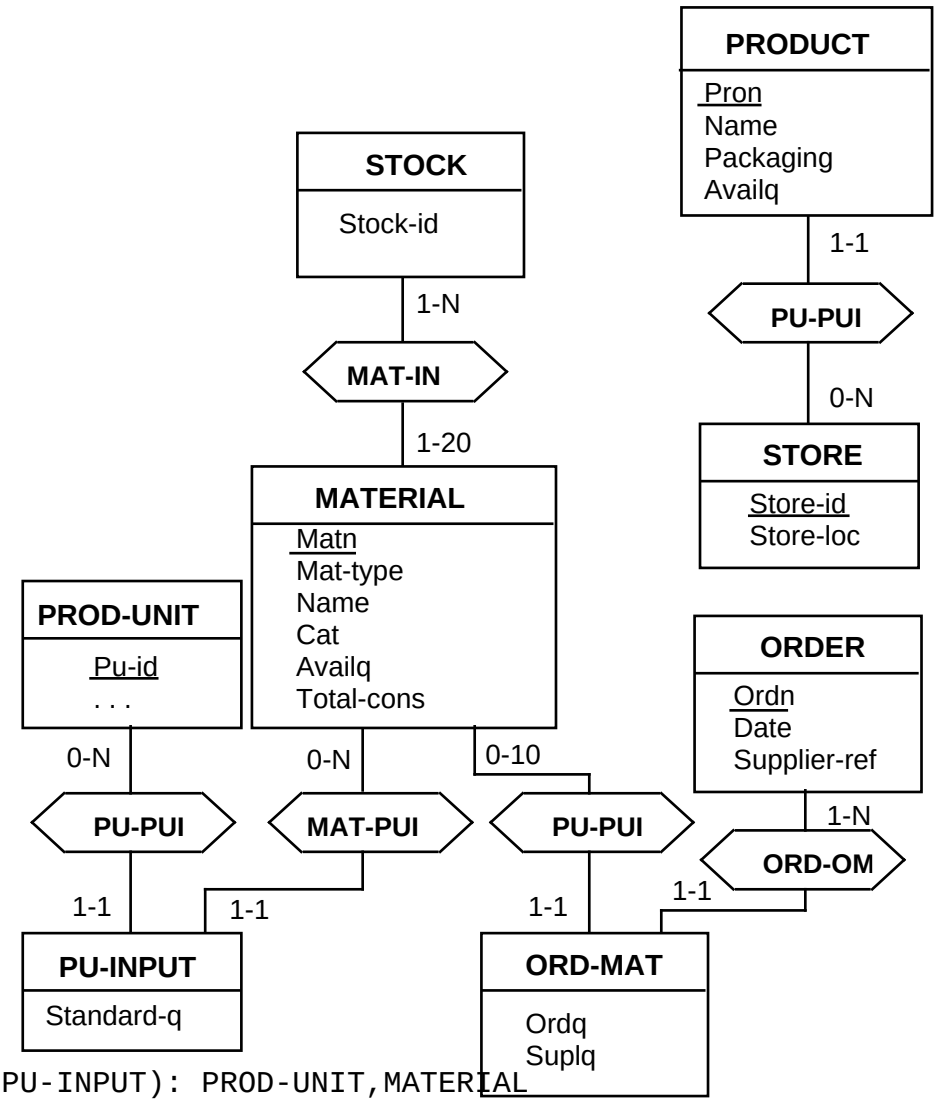
No evidences found.

RESTRUCTURATION

The MATERIAL entity type includes a multivalued attributes whose name suggests a reference to another major concept : Stock-ref. We suggest to make this concept explicit by defining entity type STOCK.



The resulting schema is a first conceptual schema, i.e. a schema that is COBOL-independent and from which all the (detected) performance-oriented constructs have been removed.



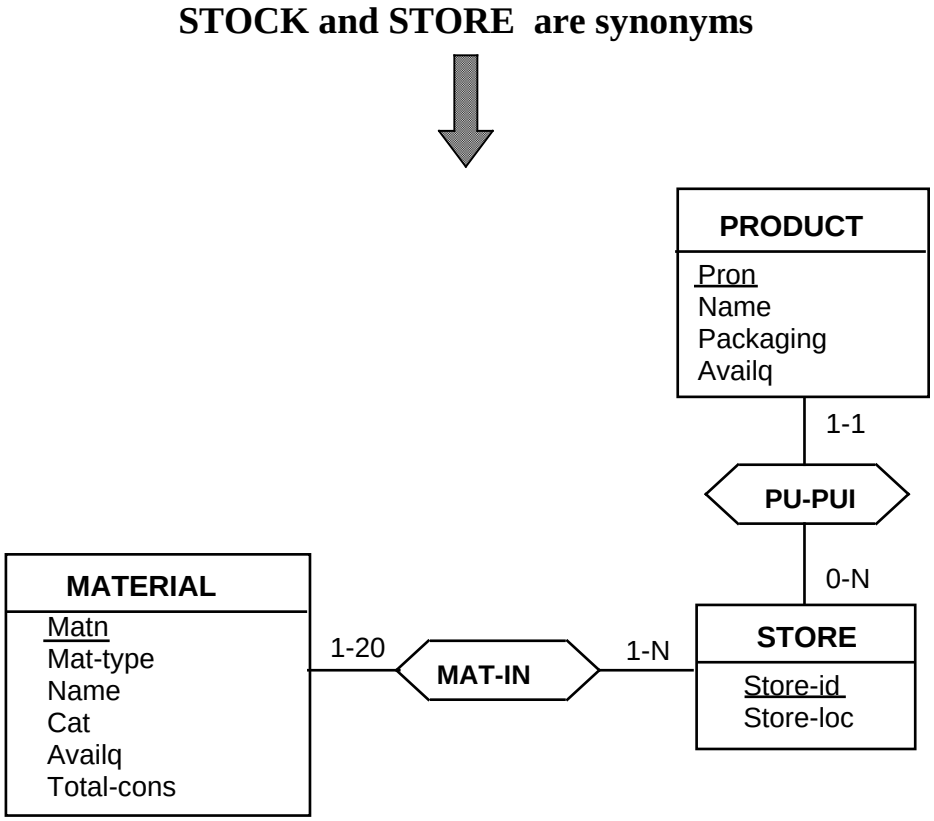
id(PU-INPUT): PROD-UNIT,MATERIAL

id(ORD-MAT):MATERIAL,ORDER

MATERIAL.Mat-type = {'R','C'}

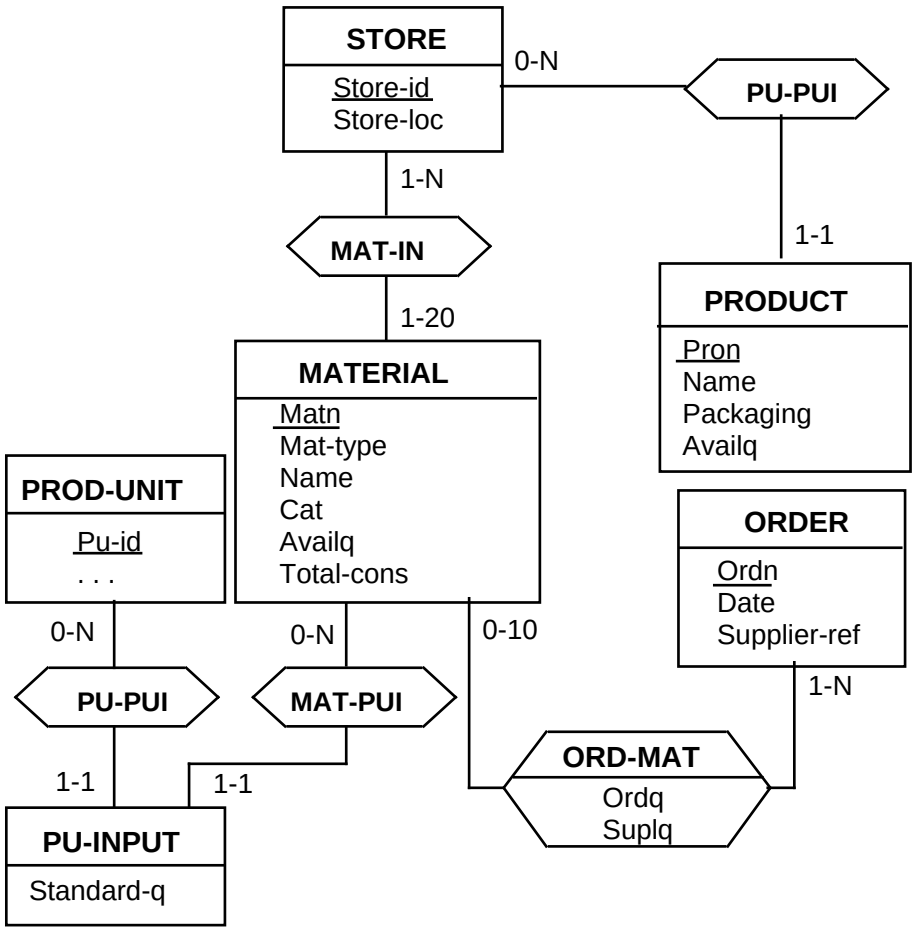
16.5.2 Conceptual normalization

The basic conceptual schema includes entity types STORE and STOCK that can be, to some extend, be considered as synonyms. This hypothesis is verified, for instance through discussion with the users. Let us suppose that these entity types expresses the same concept. The schema can be condensed :



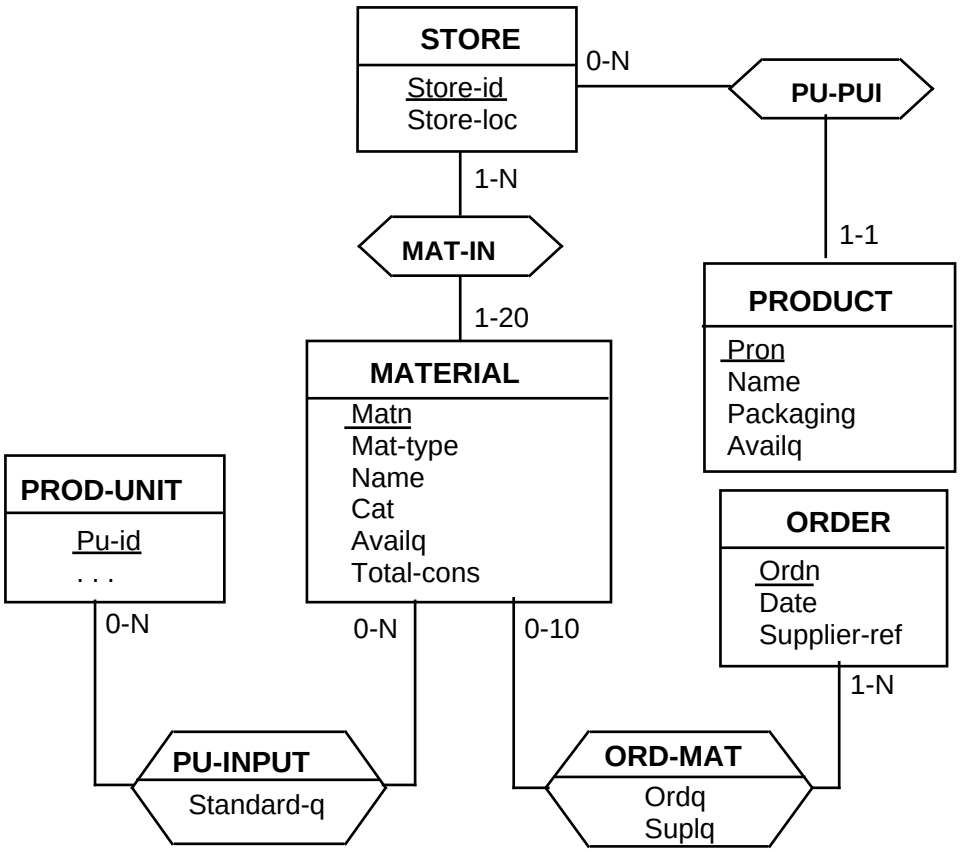
RELATIONSHIP ENTITY TYPES

Entity type ORD-MAT satisfies the preconditions for this transformation. In addition, this can be confirmed at the semantics level. Therefore, we suggest to replace it with a rel-type.



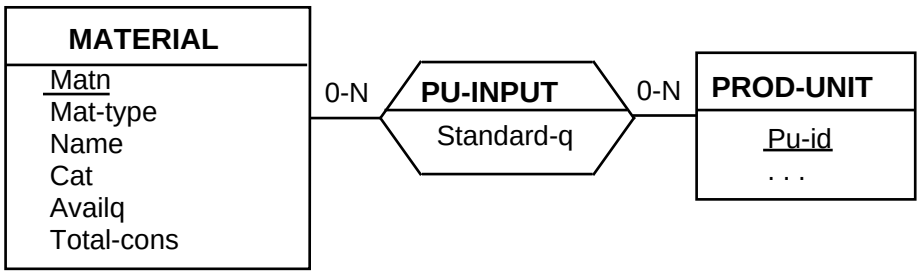

```
id(PU-INPUT): PROD-UNIT,MATERIAL
```

The same reasoning can be applied to entity type PU-INPUT, that can be replaced with a many-to-many rel-type.

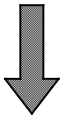


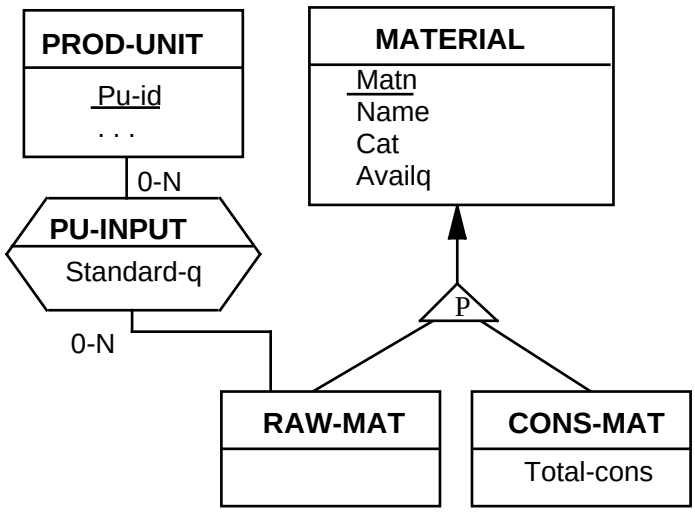
SUBTYPE STRUCTURE

The definition of (at least a subtype of) the domain of values of attribute Mat-type, its name itself (*type*) suggest a possible subtype structure. Entity type MATERIAL is examined in order to detect other evidences. There are no one-to-one rel-types, therefore exclusive components are looked for. Let us suppose that we detect such a property by file contents analysis and/or program examination. We can then propose a clearer schema :



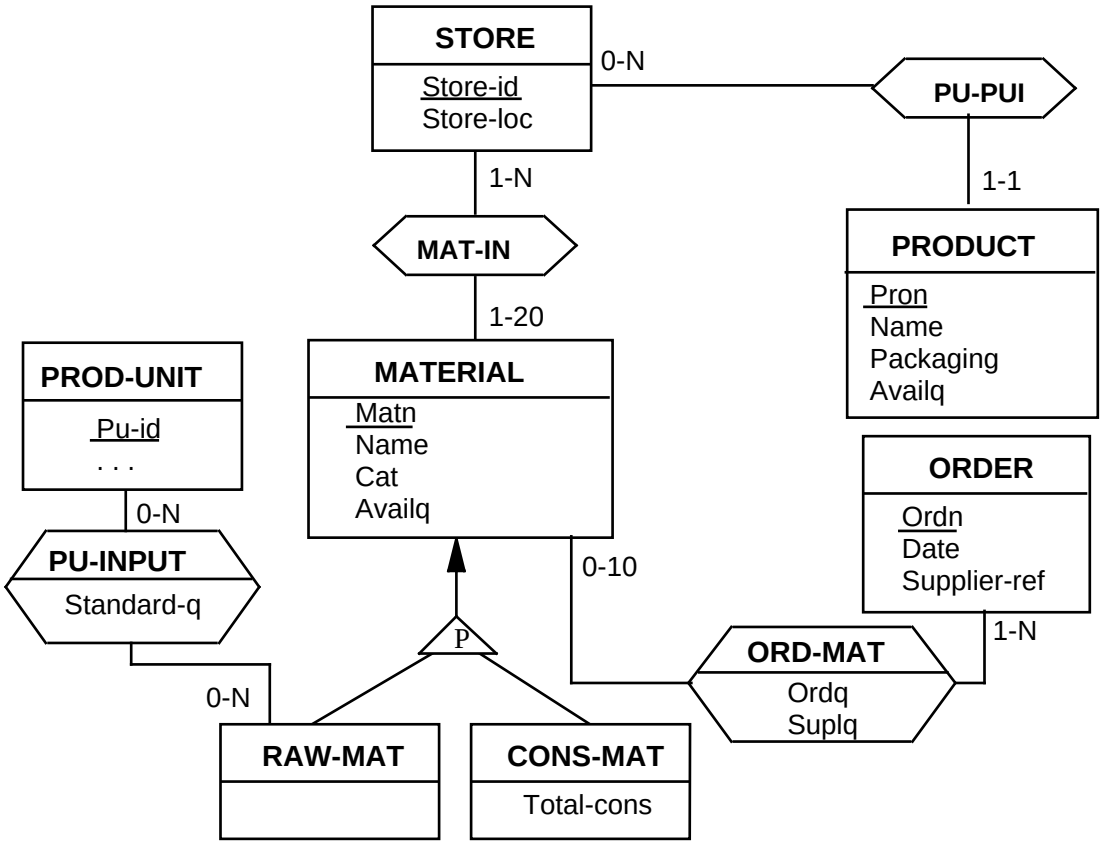

```
MATERIAL.Mat-type = {'R','C'}
excl(MATERIAL): Total-cons,PU-INPUT.MATERIAL
```





16.6 THE FINAL CONCEPTUAL SCHEMA

The conceptual schema obtained so far is as follows. As usual, it must be validated before being used for dabase documentation conversion, migration, integration, or in data administration functions.



Chapter 17

A SHORT CODASYL CASE STUDY

A third case study based on a CODASYL database description is proposed in this chapter. In addition, merging the conceptual schema that will be obtained with that of chapter 16 will be tempted to illustrate the concept of schema (and database) integration.

17.1 PRESENTATION

As in chapter 16, we will choose to work from the database declarations only. Accordingly, we will suppose that we have been provided with the SCHEMA-DDL description of the database. When needed, we will mention how hints on additional properties can be extracted from the program texts or from the database contents.

17.2 THE SOURCE CODE TEXTS

17.2.1 The Schema-DDL description

```
schema name is ORD-CUS.

record name is COMP-CUST
        location calc using cusn, duplicates not allowed.
        2  cusn pic 9(6).
        2  comp-name pic X(20).
        2  address pic X(20).
        2  category pic X(20).
        2  account pic 9(6).
        2  VAT pic 9(6).

record name is PRIV-CUST
        location calc using cusn, duplicates not allowed.
        2  cusn pic 9(6).
        2  cus-name pic X(20).
        2  chr-name pic X(15).
        2  address pic X(20).
        2  category pic X(20).
        2  account pic 9(6).

record name is PRODUCT
        location calc using pron, duplicates not allowed.
        2  pron pic 9(6).
        2  name pic X(20).
        2  unit-price pic 9(6).
        2  VAT-rate pic 9(2).

record name is ORDER
        location calc using ordnum, duplicates not allowed.
        2  ordnum pic 9(8).
        2  date pic X(6).

record name is ORDER-LINE.
        2  pronum pic 9(6).
        2  ord-q pic 9(4).
        2  suppl-q pic 9(4).
```

```
record name is INVOICE
        location calc using invnum, duplicates not allowed.
        2  invnum pic 9(6).
        2  date pic X(6).

set name PRO-OL
        order is first
        owner PRODUCT
        member ORDER-LINE mandatory automatic.

set name ORD-OL
        owner ORDER
        member ORDER-LINE mandatory automatic
        duplicates not allowed for pronum.

set name FROM-PR
        owner PRIV-CUST
        member ORDER optional manual.

set name FROM-COMP
        owner COMP-CUST
        member ORDER optional manual.

set name ORDINV
        owner ORDER
        member INVOICE mandatory automatic.

set name SYS-CUS
        order is sorted
        owner SYSTEM
        member COMP-CUST mandatory automatic
        ascending key is comp-name duplicates not allowed.
```

17.2.2 Program texts

Will be ignored in this analysis, except when mentioned otherwise.

17.3 REVERSE ENGINEERING STRATEGY

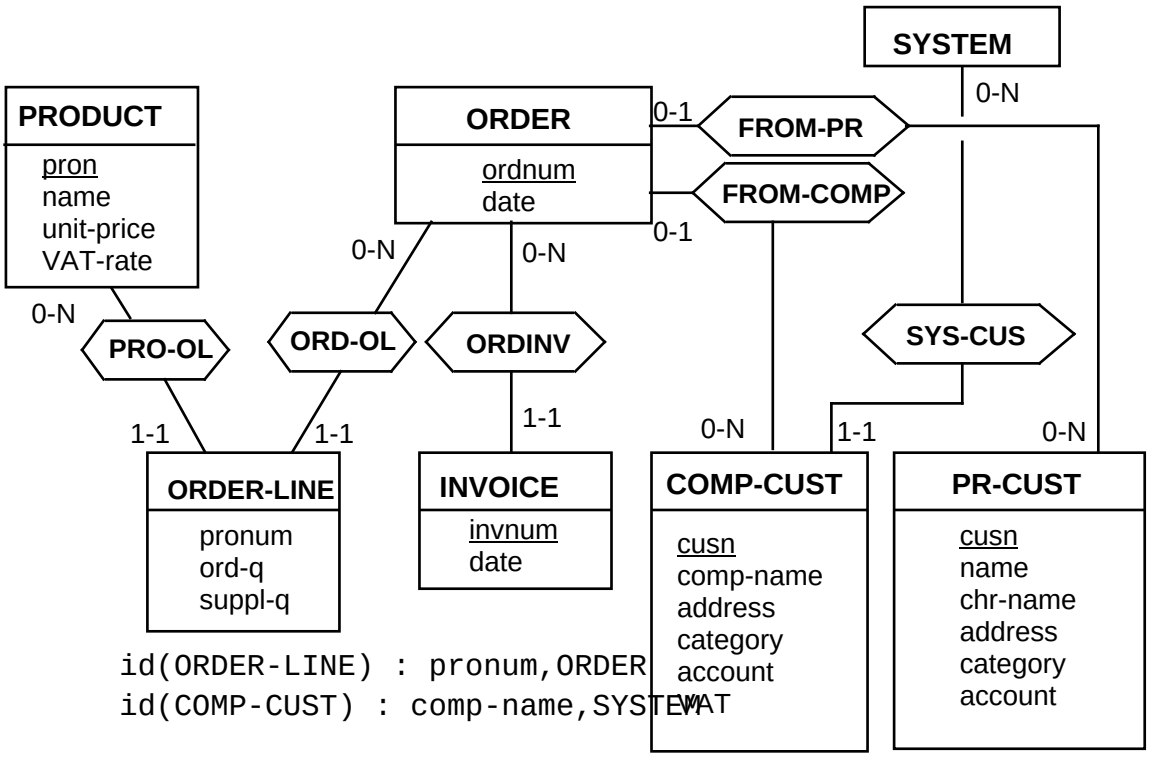
As in chapter 16, we will follow the methodology proposed in this manual. However, some flexibility will be suggested when conducting the standard processes. For instance, some activities of the two main processes will be carried out in alternately. However, we will present the example the standard way.

17.4 DATA STRUCTURE EXTRACTION

The objective of this phase is to make the RDBMS data structures and constraints explicit.

17.4.1 Global schema extraction

The SQL code is parsed and translated into data structures expressed in the ER model as follows.



Note

Examining the subschema-DDL specifications of the user's views can be useful to extract additional constraints and properties. For instance, field decomposition and mere meaningful names can be found is these descriptions.

17.4.2 Schema refinement

The procedural source code is searched for evidences of additional structures and integrity constraints.

17.5 DATA STRUCTURE CONCEPTUALIZATION

The objective of this second phase is to make the semantics of the DMS-compliant schema explicit. It consists in two major processes, namely *Basic conceptualization* and *Conceptual normalization*.

17.5.1 Basic conceptualization

Through this process, the CODASYL structures are expressed in a DMS-independent schema through the *Untranslation* process, and the optimization constructs are detected and removed through the *De-optimization* process.

17.5.1.1 CODASYL-untranslation

Due to the limited power of CODASYL in expressing complex identifiers, we will examine the schema to detect implementation of complex identifiers and of secondary identifiers.

COMPLEX IDENTIFIERS

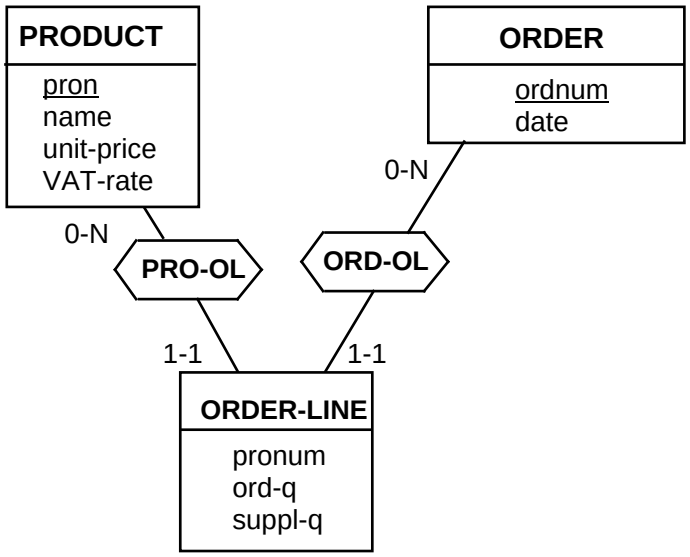
Local identifiers, i.e. data items that identifie the members of each set of a given type, can include copies of the identifier(s) of owner record type(s). In the following fragment, we can *suspect* ordnum of being a copy of the identifier of ORDER.

Hints :

- the name of the attribute
- precondition of a common practice
- analysis of the program texts

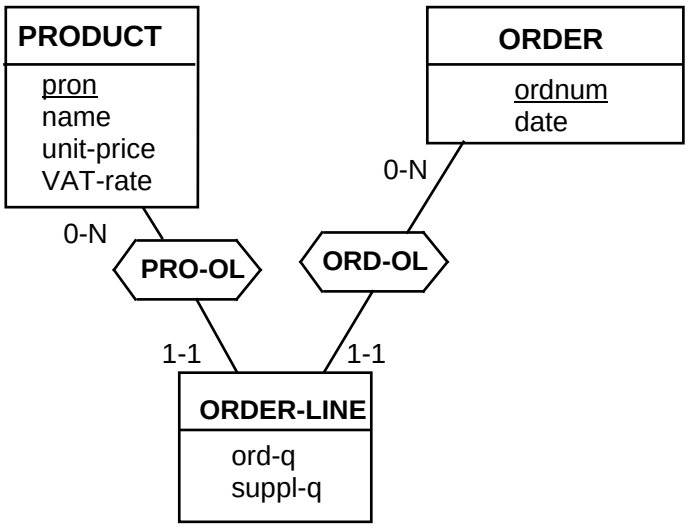
- database contents analysis.

Let us suppose that we have detected such a pattern. We augment the schema as follows :




```
id(ORDER-LINE) : pronum,ORDER
ORDER-LINE.pronum copy-of ORDER-LINE.PRO-OL.PRONU
```

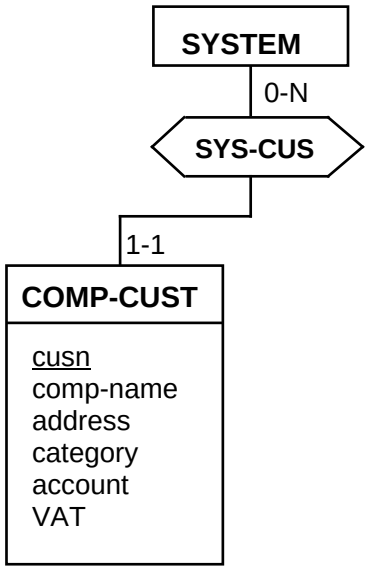
We are now in a position to simplify this fragment as follows :




```
id(ORDER-LINE) : PRODUCT,ORDER
```

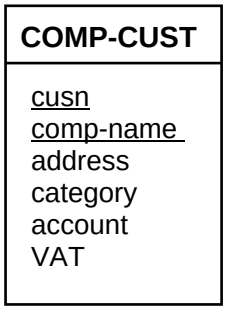
SECONDARY IDENTIFIERS

Secondary identifiers, i.e. data items that identifie records of a given type and that are not part of the CALC key, are generally implemented as SYSTEM set local identifiers. The schema include such a construct.




```
id(COMP-CUST) : comp-name,SYSTEM
```

When expressing this pseudo-local identifier as an absolute identifier, we have to examine the other roles of the SYSTEM entity type. In this case, it is no longer linked to any other entity type. Therefore, it can be removed from the schema. Hence the following fragment.



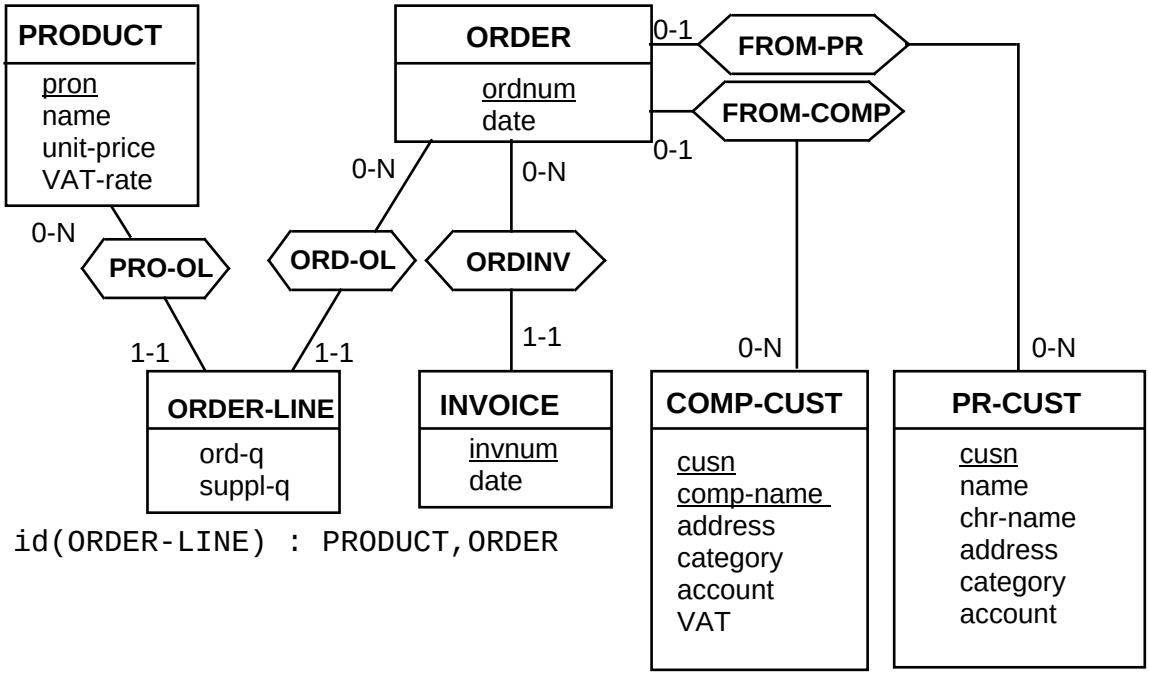
REFERENCE ATTRIBUTES

Many CODASYL databases includes reference attributes (some kind of *foreign keys*). No such constructs are detected in this schema.

17.5.1.2 De-optimization

The resulting schema is then analysed in order to detect possible traces of optimization constructs. No traces are found in this schema

The resulting schema is a first conceptual schema, i.e. a schema that is DMS-independent and from which all the (detected) performance-oriented constructs have been removed.



17.5.2 Conceptual normalization

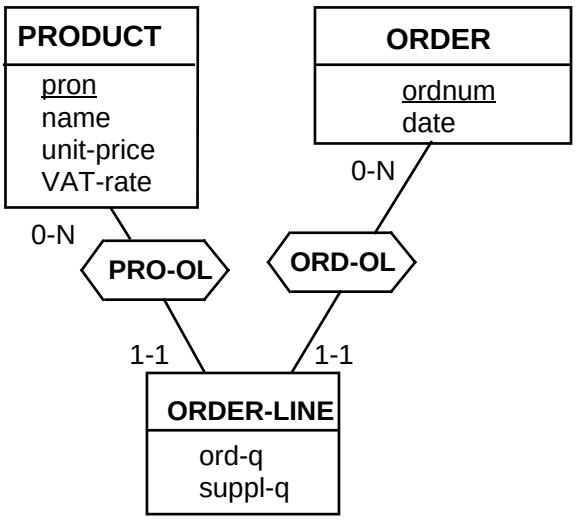
This first conceptual schema can be restructured in order to give it a higher degree of clarity, minimality and self-expressiveness. We will examine some common structures that can be expressed in a clearer way.

ATTRIBUTE ENTITY TYPES

No evidences.

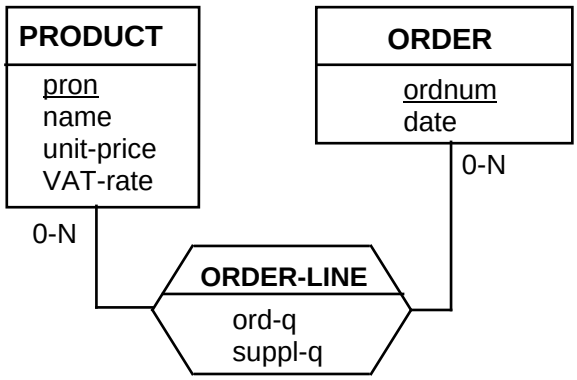
RELATIONSHIP ENTITY TYPES

Entity type ORDER-LINE seems to fall in this category :



`id(ORDER-LINE) : PRODUCT,ORDER`

We choose to replace it with a rel-type between PRODUCT and ORDER :



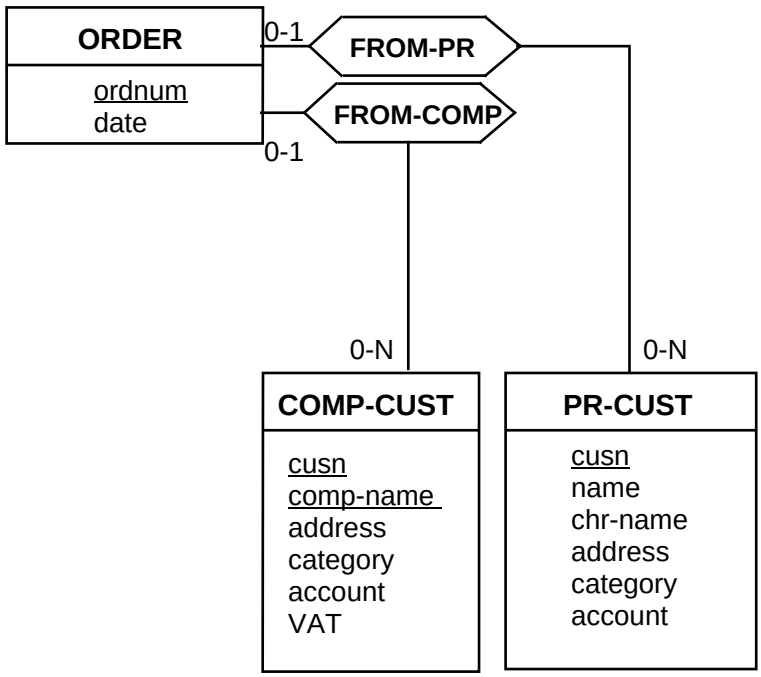
There are no other similar pattern.

SUBTYPE STRUCTURE

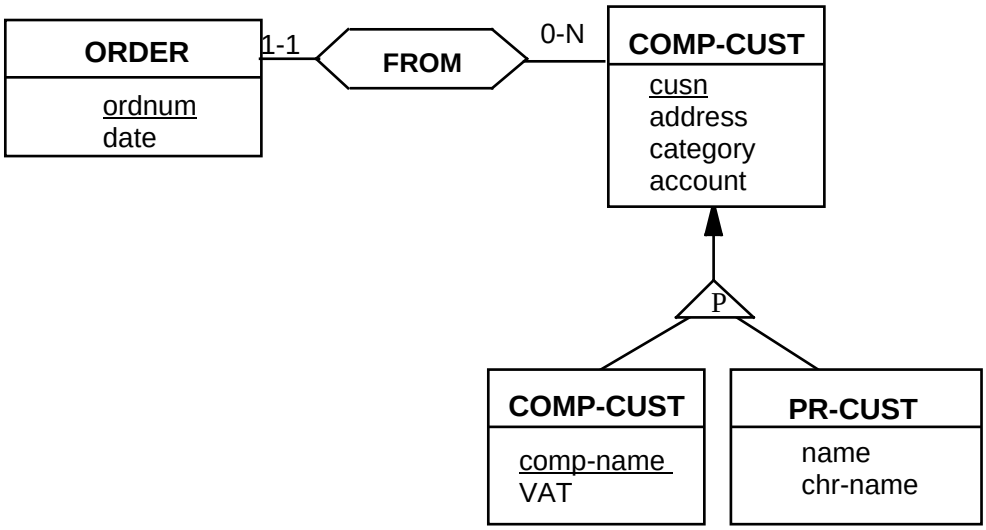
Due to the following hints :

- the similarity of the names of COMP-CUST and PR-CUST
- the apparent common attributes
- the similarities of rel-types FROM-PR and FROM COMP
- the similarities of the names of these entity types and rel-types,

we can guess that these entity types could be subtypes of a common super type to discover. To go a bit further, we examine possible global identifiers and exclusion constraints. Let us suppose that we have succedded in proving the following properties :

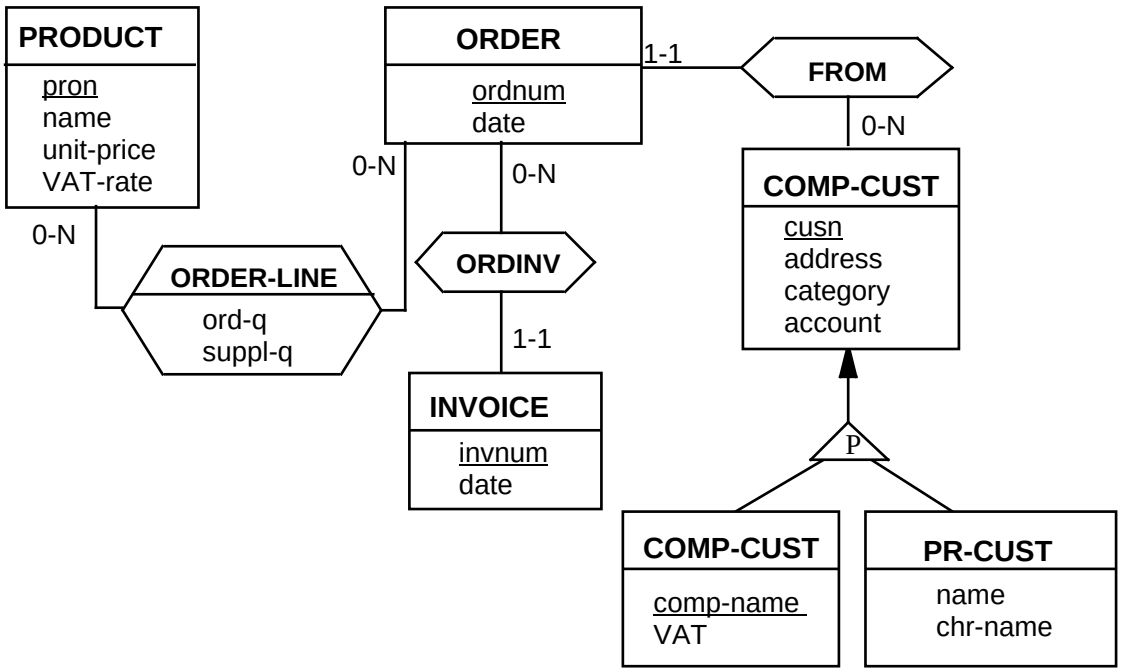


We can then propose a simpler expression of this schema :



17.6 THE FINAL CONCEPTUAL SCHEMA

The conceptual schema obtained so far should be validated against user's perception. It could appear as follows.



17.7 INTEGRATING TWO DATABASES

Let us suppose that this CODASYL database is used in the same company as tha COBOL files processed in chapter 16. In addition, let us suppose that both databases have to be integrated in some ways. We are now faced with a problem of conceptual view integration.

In this case, the problem is rather simple. Acording to the semantics of the schemas, two correspondences can be suggested :

- COBOL ORDER = CODASYL ORDER ?
- COBOL PRODUCT = CODASYL PRODUCT ?

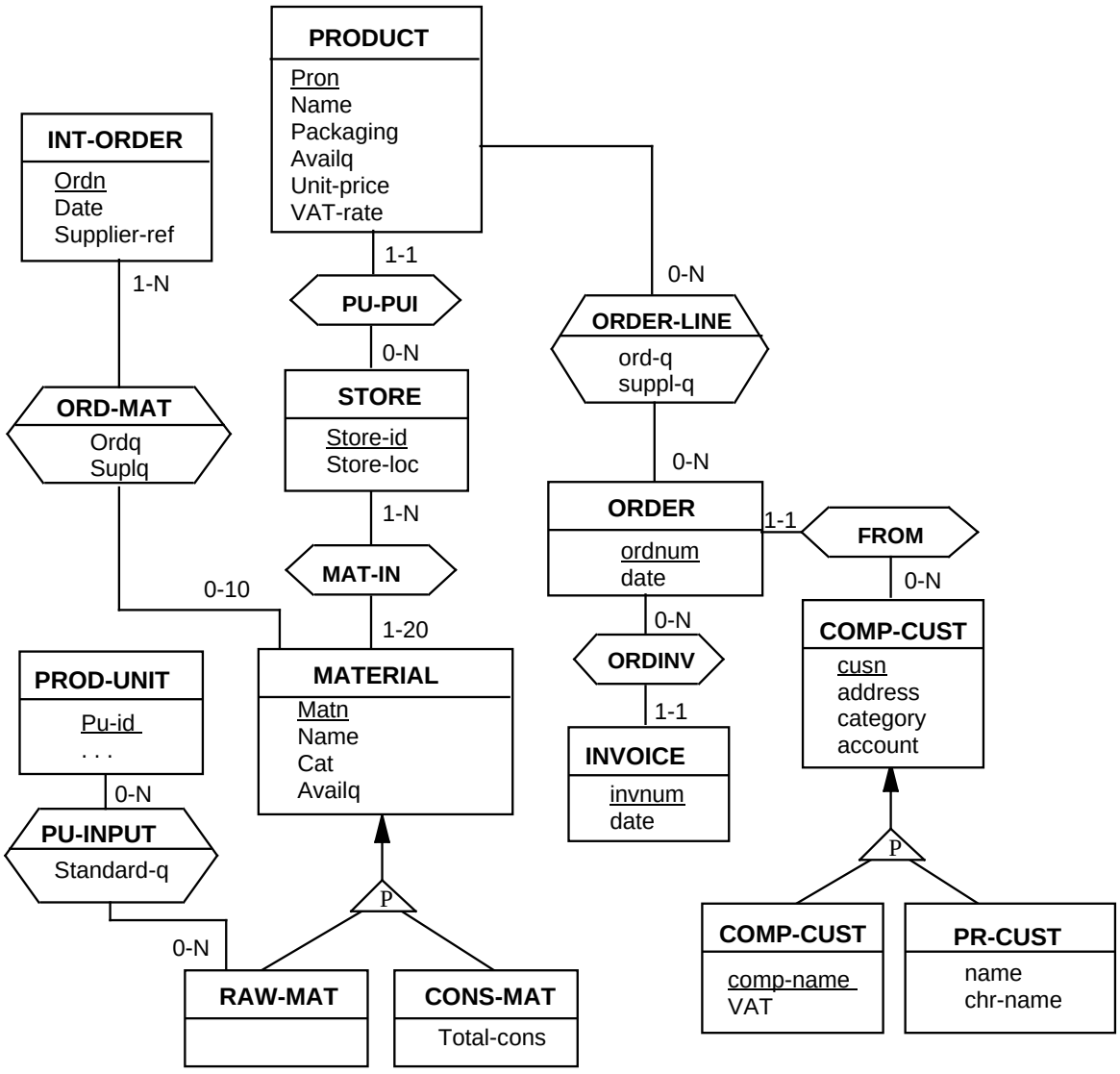
Let us examine each of them.

It appears that the COBOL ORDER concerns ordering materials used internally, while CODASYL ORDER are those passed by the external customers. These orders are processes in quite different ways. Therefore, they must be kept seperate.

The situation for PRODUCT is different. COBOL PRODUCT represents the products manufactured by the company, while the CODASYL PRODUCT represents the products that the compny sells to customers. These products are the same. Therefore, their entity

types can be merged. This integration implies integrating the attributes and the roles of the entity types as well.

The integrated schema appears as follows :



Chapter 18

REFERENCES

ANSI,75

ANSI/X3/SPARC, "Study Group on Data Base Management System interim Report", in FDT (ACM-SIGMOD) Bulletin, Vol. 7, No 2, 1975

BACHMAN,69

Bachman, C., "Data Structure Diagrams", in Data Base, Vol. 1, No 2, 1969

BATINI,83

Batini, C., Lenzerini, M., Moscarini, M., *View integration, in Methodology and tools for data base design*, Ceri, S., (Ed.)North-Holland, 1983

BATINI,86

Batini, C., Lenzerini, M., Navathe, S., B, *A comparative Analysis of Methodologies for Database Schema Integration*, in ACM Computing Survey, Vol. 15, No 4, pp. 323-364, December, 1986

BATINI,92

Batini, C., Ceri, S., Navathe, S., B., *Conceptual Database Design*, Benjamin/ Cummings, 1992

BEERI,89

Beeri, C., "Formal Methods for Object Oriented Databases", in Proc. of Deductive and Object Oriented Database Systems, 1989

BERT,85

Bert, M., N., al., *The logical design in the DATAID project : the EASYMAP system*, in Computer-Aided Database Design : the DATAID Project, Albano, al., (Ed.)North-Holland, 1985

BODPIG,89

Bodart, F., Pigneur, Y., "Conception assiste des systmes d'information - Mthodes, modles et outils (2e edition)", Masson, 1989

BOULANGER,89

Boulanger, D., March, S. T., "An approach to analyzing the information content of existing databases", in Data Base, Vol. , No , Summer, 1989

BOUZHEGOUB,90

Bouzheghoub, M., Comyn-Wattiau, I., *View Integration by Semantic Unification and Transformation of Data Structures*, in Proc. of 9th Entity-Relationship Approach, 1990

BRAGGER,84

Brägger, R., P., Dudler, A., Rebsamen, J., Zehnder, C., A., *GAMBIT : An Interactive Database Design Tool for Data Structures, Integrity Constraints, and Transaction*, in Proc. of Data Engineering, pp. 399-407,1984

CASANOVA,83

Casanova, M., Amarel de Sa, J., *Designing Entity Relationship Schemas for Conventional Information Systems*, in Proc. of Entity-Relationship Approach, pp. 265-278, 1983

CASANOVA,84

Casanova, M., A., Amaral De Sa, *Mapping uninterpreted Schemes into Entity-Relationship diagrams : two applications to conceptual schema design*, in IBM J. Res. & Develop., Vol. 28, No 1, January, 1984

CASEY,73

Casey, R., G., Delobel, C., "Decomposition of Data Base and the Theory of Boolean Switching Functions", in IBM J. of R & D, Vol. 17, No 5, September, 1973

CHEN,76

Chen, P., P., *The Entity-Relationship Model - Towards a Unified View of Data*, in ACM TODS, Vol. 1, No 1, pp. 9-36, , 1976

CHIKOFSKY,90

Chikofsky, E. J., Cross II, J. H., "Reverse Engineering and Design Recovery : A Taxonomy", in IEEE Software, Vol. , No , January, 1990

CODD,79

Codd, E., F., "Extending the Database Relational Model to Capture more Meaning", in ACM TODS, Vol. 4, No 4, December, 1979

COLLART,88

Collart, N., Joris, M., "Etude théorique et pratique d'extensions du modle Entité-Association", Master thesis , Institut d'Informatique - FUNDP, September 1988

CURTIS,92

Curtis, B., Kellner, M., I., Over, J., *Process Modeling*, Comm. of the ACM, Vol. 35, No. 9, pp.75-90, Sept. 1992

DATAID,83

Methodology and tools for data base design, Ceri, S., (Ed.), North-Holland, 1983

DATAID,85

Computer-Aided Database Design : the DATAID Project, Albano, al., (Ed.), North-Holland, 1985

DATE,86

Date, C., J., "An Introduction to Database Systems (Volume I)", Addison-Wesley, 1986

DATRI,84

D'Atri, A., Sacca', D., *Equivalence and Mapping of Database Schemes*, in Proc. of Very Large Databases, pp. 187-195, August, 1984

DAVIS,85

Davis, K., H., Adarsh, K., A., *A Methodology for Translating a Conventional File System into an Entity-Relationship Model*, in Proc. of Entity-Relationship Approach, Octobre, 1985

DAVIS,88

Davis, K., H., Arora, A., K., *Converting a Relational Database model to an Entity Relationship Model*, in Proc. of Entity-Relationship Approach : a Bridge to the User, 1988

DEHENEFFE,74

Deheneffe, C., Hainaut, J.L., Tardieu, H., "The individual model", in Proc. of International Workshop on Data Structure Models for Information Systems, 1974

DUBOIS,87

Dubois, E., Van Lamsweerde, A., *Making Specification Processes*, in Proc. of the 4th Intern. Workshop on Software Specification and Design, Monterrey (CA), April, 3-4, 1987, pp. 169-177

ELMASRI,89

Elmasri, R., Navathe, S., *Fundamentals of Database Systems*, Benjamin/Cummings, 1989

FAGIN,77

Fagin, R., "Multivalued dependencies and a new normal form for relational databases", in ACM TODS, Vol. 2, No 3, , 1977

FONKAM,92

Fonkam, M., M., Gray, W., A., *An approach to Eliciting the Semantics of Relational Databases*, in Proc. of 4th Int. Conf. on Advance Information Systems Engineering - CAiSE'92, pp. 463-480, May, LNCS, Springer-Verlag, 1992

GALLAIRE,84

Gallaire, H., Minker, J., Nicolas, J.M., "Logic and Databases : a deductive approach", in ACM Comp. Surveys, Vol. 16, No 2, June, 1984

GELLER,89

Geller, J., R., *IMS, Administration, Programming and Data Base Design*, Wiley, 1989

GIRAUDIN,85

Giraudin, J-P., Delobel, C., Dardailler, P., *Eléments de construction d'un système expert pour la modélisation progressive d'une base de données,* in Proc. of Journées Bases de Données Avancées, Mars, 1985

HAINAUT,81

Hainaut, J-L., *Theoretical and practical tools for data base design*, in Proc. of the Very Large Databases Conf., pp. 216-224, September, 1981

HAINAUT,86

Hainaut, J-L., *Conception assistée des applications informatiques - 2. Conception de la base de données*, Masson, 1986

HAINAUT,89

Hainaut, J.-L., *A Generic Entity-Relationship Model*, in Proc. of the IFIP WG 8.1 Conf. on *Information System Concepts: an in-depth analysis*, North-Holland, 1989

HAINAUT,90

Hainaut, J-L., *Entity-Relationship models : formal specification and comparison - Tutorial*, in Proc. of Entity-Relationship Approach : the Core of Conceptual Modelling, pp. 433-444, North-Holland, 1991

HAINAUT,91a

Hainaut, J-L., *Entity-generating Schema Transformation for Entity-Relationship Models*, in Proc. of the 10th Entity-Relationship Approach, San Mateo (CA), 1991

HAINAUT,91b

Hainaut, J-L, *Database Reverse Engineering, Models, Techniques and Strategies*, in Proc. of the 10th Conf. on Entity-Relationship Approach, San Mateo (CA), 1991

HAINAUT,92a

Hainaut, J-L., Cadelli, M., Decuyper, B., Marchand, O., *Database CASE Tool Architecture : Principles for Flexible Design Strategies*, in Proc. of the 4th Int. Conf. on Advanced Information System Engineering (CAiSE-92), Manchester, May 1992, Springer-Verlag, LNCS, 1992

HAINAUT,92b

Hainaut, J-L., *A Temporal Statistical Model for Entity-Relationship Schemas*, in Proc. of the 11th Conf. on the Entity-Relationship Approach, Karlsruhe, Oct. 1992, Springer-Verlag, LNCS, 1992

HAINAUT,92c

Hainaut, J-L., Cadelli, M., Decuyper, B., Marchand, O., *TRAMIS : a transformation-base database CASE tool*, in Proc. 5th Int. Conf. on Software Engineering and Applications, Toulouse, 7-11 December, 1992

HAINAUT,93a

Hainaut, J-L., Chandelon M., Tonneau C., Joris M., *Contribution to a Theory of Database Reverse Engineering*, in Proc. of the IEEE Working Conf. on Reverse Engineering, Baltimore, IEEE Computer Society Press, May 1993

HAINAUT,93b

Hainaut, J-L, *Schema Transformation for Database Engineering - Theoretical Elements*, Research report, Institut d'Informatique, FUNDP, Namur, May, 1993

HAINAUT,93c

Hainaut, J-L., Tonneau C., Joris M., Chandelon M., *Schema Transformation Techniques for Database Reverse Engineering*, in Proc. of the 12th Int. Conf. on Entity-Relationship Approach, Arlington-Dallas, December 1993 (to be published).

HALL,92

Software Reuse and Reverse Engineering in Practice, Hall, P., A., V. (Ed.), Chapman&Hall, 1992

IEEE,90

Special issue on Reverse Engineering, IEEE Software, January, 1990

JACOBSON,91

Jacobson, I., Lindström, F., *Re-engineering of old systems to an object-oriented architecture*, in Proc of OOPSLA'91, pp.340-350, 1991

JARDINE,89

Jardine, D., A., Yazid, S., *Integration of Information Submodels*, in Proc. of Information System Concepts : An In-depth Analysis, pp. 247-267, October, 1989

JORIS,92

Joris, M., Van Hoe, R., Hainaut, J-L., Chandelon M., Tonneau C., Bodart F. et al., *PHENIX : methods and tools for database reverse engineering*, in Proc. 5th Int. Conf. on Software Engineering and Applications, Toulouse, 7-11 December, 1992

KOBAYASHI,86

Kobayashi, I., *Losslessness and Semantic Correctness of Database Schema Transformation : another look of Schema Equivalence*, in Information Systems, Vol. 11, No 1, pp. 41-59, January, 1986

KRASNER,92

Krasner, H., Terrel, J., Linehan, A., Arnold, P., Ett, W., H., *Lessons learned from a software Process Modeling System*, Comm. of the ACM, Vol. 35, No. 9, pp.91-100, Sept. 1992

KRIEG,89

Krieg-Brückner, B., *Algebraic Specification and Functionals for Transformational Program and Meta Program Development*, in Proc. of the TAPSOFT Conf. LNCS 352, Springer-Verlag, 1989

KOZACZYNSKY,87

Kozaczynsky, Lilien, *An extended Entity-Relationship (E2R) database specification and its automatic verification and transformation*, in Proc. of Entity-Relationship Approach, 1987

LARSON,89

Larson, J.A., Navathe,S.B., Elmasri,R., "A Theory of Attribute Equivalence in Databases with Application to Schema Integration", in IEEE Transactions on Software Engineering, Vol. 15, No 4, April, 1989

LING,89

Ling, T., W., *External schemas of Entity-Relationship based DBMS*, in Proc. of Entity-Relationship Approach : a Bridge to the User, 1989

MATHERON,91

Matheron, J.P., *Approfondir MERISE*, Eyrolles, Paris, 1991

METAXIDES,75

Metaxides, A., *"Information bearing" and "non-information bearing" sets*, in Data Base Description, IFIP TC2 Work. Conf., Douqué & Nijssen (Ed.), pp363-368, North-Holland, 1975

MOTRO,87

Motro, *Superviews : Virtual Integration of Multiple Databases*, in IEEE Trans. on Soft. Eng., Vol. SE-13, No 7, July, 1987

NAVATHE,80

Navathe, S., B., *Schema Analysis for Database Restructuring*, in ACM TODS, Vol.5, No.2, June 1980

NAVATHE,84

Navathe, S., B., Sashidar, T., Elmasri, R., A., *Relationship Merging in Schema Integration*, in Proc. of Very Large Databases, 1986

NAVATHE,88

Navathe, S., B., Awong, A., *Abstracting Relational and Hierarchical Data with a Semantic Data Model*, in Proc. of Entity-Relationship Approach : a Bridge to the User, 1988

NILSSON,85

Nilsson,E., G., *The Translation of COBOL Data Structure to an Entity-Rel-type Conceptual Schema*, in Proc. of Entity-Relationship Approach, October, 1985

OLLE,82

Information Systems Design Methodologies : a Comparative Review, Olle, W., Sol, H., Verrijn-Stuart, A., (Ed.), North-Holland, 1982

OVUM,90

Rock-Evans, R., "Reverse Engineering : Markets, Methods and Tools", OVUM report, 1990

PARTSH,83

Partsch, H., Steinbrüggen, R., *Program Transformation Systems*, Computing Surveys, Vol. 15, No. 3, 1983

PERRON,81

Perron, *Design Guide for CODASYL DBMS*, QED Information Sciences, 1981

REINER,86

Reiner, D., Brown, G., Friedell, M., Lehman, J., McKee, R., Rheingans, P., Rosenthal, A., *A Database Designer's Worbench*, in Proc. of Entity-Relationship Approach, 1986

ROCK,90

Rock-Evans, R., *Reverse Engineering : Markets, Methods and Tools*, OVUM report, 1990

ROSENTHAL,88
Rosenthal, A., Reiner, D., *Theoretically sound transformations for practical database design*, in Proc. of Entity-Relationship Approach, 1988

SCHIEL,84
Schiel, U., Furtado, A., L., Neuhold, E., J., Casanova, M., A., *Towards Multi-level and Modular Conceptual Schema Specifications*, in Information Systems, Vol. 9, No 1, pp. 43-57, 1984

SENKO,73
Senko, M., Altmon, Astrakan, Fehder, "Data Structures and Accessing in Data-Base Systems", in IBM Syst. J., Vol. 12, No 1, 1973

SHIPMAN,81
Shipman, D., "The Functional Data Model and the Data Language DAPLEX", in ACM TODS, Vol. 6, No 1, March, 1981

SOWA,84
Sowa, "Conceptual Structures - Information Processing in Mind and Machine", Addison-Wesley, 1984

SPACCA,90a
Spaccapietra, S., Parent, C., "View Integration : A Step Forward in Solving Structural Conflicts", Res. Report , EPFL, Lausanne (CH), August 1990

SPACCA,91
Spaccapietra, S., Parent, C., Dupont, Y., *Automating Heterogeneous Schema Integration*, EPFL, Lausanne (CH), February, 1991

SPACCA,92
Spaccapietra, S., Parent, C., *View Integration : A Step Forward in Solving Structural Conflicts*, Res. Report , EPFL, Lausanne (CH), IEEE Trans. on Knowledge and Data Engineering, October, 1992

SPRING,90
Springsteel, F., N., Kou, C., *Reverse Data Engineering of E-R designed Relational schemas*, in Proc. of Databases, Parallel Architectures and their Applications, March, 1990

STOREY,90
Storey, V. C., Goldstein, R. C., "Some Findings on the Intuitiveness of Entity-Relationship Constructs", in Proc. of 8th Int. Conf. on Entity-Relationship Approach, Lochovsky editor, pp. 9-23, 1990

TARDIEU,83
Tardieu, H., Rochfeld, Coletti, *La méthode Merise*, Les Editions d'Organisation, 1983

TEOREY,86

Teorey, T., J., Yang, D., Fry, J., P., *A Logical Design Methodology for Relational Databases using the Extended Entity-Relationship Model*, in ACM Comp. Surveys, Vol. 18, 2, No , pp. 197-222, 1986

TEOREY,89

Teorey, T. J., *Database design*, Prentice-Hall, 1989

TRAMIS,90

Cadelli, M., Decuyper, B., Hainaut, J.L., "TRAMIS : a transformation-based database CASE tool", Technical report , Institut d'Informatique - Namur, September 1990

ULLMAN,89

Ullman, J., D., *Principles of Data- and Knowledge-base Systems* (Vol I & II), Computer Science Press, 1989

VERHEIJEN,82

Verheijen, Van Bekkum, "NIAM : an Information Analysis Method", in [OLLE,82]

WINANS,90

Winans, J., Davis, K., H., *Software Reverse Engineering from a Currently Existing IMS Database to an Entity-Relationship Model*, in Proc. of Entity-Relationship Approach : the Core of Conceptual Modelling, pp. 345-360, October, 1990

WOODS,75

Woods, "What's in a link : foundation for semantic networks", in "Representation and Understanding", Academic Press, 1975