Case study **30**

Classifying objects

Objective. In this study, we explore a particular way of classifying objects based on their attributes. This technique, called *Formal Concept Analysis*, or *FCA* for short, examines the composition of these objects and extracts *concepts*, that is, classes of objects that share the same set of attributes. By considering the inclusion relationship of the concept object sets, the concepts can be organized as a hierarchy.

Several techniques have been designed to extract concepts from a set of source objects and to build their hierarchy. We analyze the reasoning underlying these techniques and we develop one of the most popular of them, the Chein algorithm. We first translate this iterative algorithm into a Python procedure then we express it as an SQL script.

We propose a third, much simpler and faster technique that produces a remarkable subset of the Chein concept hierarchy. It appears that this technique, which can be coded as a single SQL query or in a small Python procedure, is more appropriate to database schema processing, specifically to conceptual schema normalization and to reverse engineering legacy databases.

The study develops four parametric applications to experiment with these algorithms and to evaluate their performance in time and space.

Keywords. symbolic classification, Formal Concept Analysis (FCA), Galois lattice, set operators, performance evaluation, database optimization, algorithm optimization

Table of content

30.1 Introduction and motivation

Let us first read Wikipedia on the main subject of this study [Wikipedia, 2023]:

> *Formal concept analysis (FCA) is a principled way of deriving a concept hierarchy or formal ontology from a collection of objects and their properties. Each concept in the hierarchy represents the objects sharing some set of properties; and each sub-concept in the hierarchy represents a subset of the objects (as well as a superset of the properties) in the concepts above it. The term was introduced by Rudolf Wille in 1981, and builds on the mathematical theory of lattices and ordered sets that was developed by Garrett Birkhoff and others in the 1930s.*
>
> *Formal concept analysis finds practical application in fields including data mining, text mining, machine learning, knowledge management, semantic web, software development, chemistry and biology.*

The technique we will examine in this case study, called *Formal Concept Analysis*, or *FCA*, aims at extracting pertinent concepts from large data sets. Contrary to statistical analysis, it does not rely on numerical computing, such as the extraction of average and standard deviation, or regression analysis. Rather, it applies symbolic transformations on simple graphs, based on the mathematical theory of *Galois lattice* [Huchard, 2000].

We will explore this concept in various application domains related to knowledge extraction and processing, notably structuring large sets of elementary objects and database schema manipulation[1].

A short illustration of FCA

We will illustrate FCA by one of its historical applications: eliciting *subtype/supertype* hierarchy in a database schema.

The basic idea is that, when two or more entity types appear to have attributes in common, these common components can be extracted to form a new entity type, that is declared a supertype of the former.

Let us have a look at the schema of Figure 30.1 that describes two data sets, named ENGINEER and SECRETARY. The very nature of these data sets is irrelevant in this presentation: they can be structured data files, tables in a standard relational database, table types in SQL3 object-relational databases, document types, Java classes, object classes of an ontology or entity types of a database we intend to develop (what one generally calls its *conceptual schema*). Let us call them **concepts**, in that they denote abstract categories of concrete objects.

1. The first domain concerns untyped instances (such as messages) for which we try to derive the most relevant type(s). In the second domain, we deal with objects that are already types, of which we ignore the instances. Only the first level of abstraction distinguishes these two domains.

To the readers who may feel uncomfortable with this idea of *abstraction*, we suggest to consider Figure 30.1 as the schema of a database comprising two tables. The data rows are the concrete objects while the schema represents their abstraction.

We consider that the concept named ENGINEER represents the class of employees of a company in charge of identifying and solving technical problems. Each of them is characterized by an *employee Id* (represented by attribute *Emp#*), a *name*, a skill *level*, a *salary* and a *specialty*. The other concept is that of SECRETARY, designating the class of the employees in charge of administrative and logistical tasks. Each secretary has a similar set of attributes: *employee Id*, *name*, *salary* and the *language* in which they are most fluent.

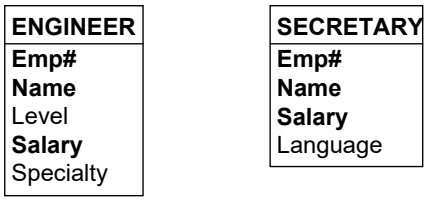

Figure 30.1 - An elementary source schema: two concepts have some attributes in common

The property of this schema we are interested in is that these concepts *share common attributes*, namely Emp#, Name and Salary (in bold face in Figure 30.1). This naturally suggests that all the engineers and all the secretaries are also instances of a more general concept the attributes of which are these common attributes. We should assign a name to this new concept. It seems natural to call *Employees* the category of objects

– that all have an *employee number*, a *name* and a *salary*?

– and that comprises two subcategories, *engineers* and *secretaries*?

Hence the concept EMPLOYEE.

Now, our schema comprises three concepts, EMPLOYEE, ENGINEER and SECRETARY. EMPLOYEE is more general than both ENGINEER and SECRETARY. Conversely, the latter are more specific than EMPLOYEE. They often are called the **subconcepts** of EMPLOYEE, while the latter is the **superconcept** of ENGINEER and SECRETARY.

The new schema is represented in Figure 30.2. Considering the concrete level, it states that,

– each engineer is also an employee

– each secretary is also an employee

and, conversely,

– some employees are engineers

– some employees are secretaries.[2]

A star structure (a triangle plus three branches) connects the general concept (the thick branch) with its specific concepts (the thin branches). As to the attributes, we have **migrated** the common ones to the general concepts while the attributes specific to a concept are kept with it.

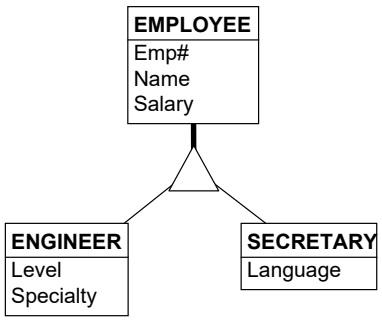

Figure 30.2 - Extracting common attributes as a new concept

This schema must be read as follows (example of ENGINEER):

– ENGINEER has five attributes,

– among them, two are **specific** (also called **proper**) attributes: Level and Specialty,

– and three are **inherited** attributes: Emp#, Name and Salary.

Usually, one represents only the proper attributes of a concept, as shown in the schema of Figure 30.2. We could also have chosen to show all the attributes, proper and inherited, of each concept but this would have made the schema less readable. In a sense, each attribute is located where it is the most relevant.

Four important observations to keep in mind for the following discussion:

– The restructuring operation creates a *hierarchy* of concepts.

– This restructuring operation is only valid if attributes with the same name in the specific concepts convey the same meaning. Should SECRETARY attribute Language be named Specialty, the resulting schema would have been quite different and most probably less informative.

– Being based on purely symbolic (more precisely *character string*) manipulations, this operation can be automated[3].

– In set theoretic terms, the new concept is built as the *union* of existing concepts and its attributes are the *intersection* of those of the existing concepts.

2. If Figure 30.2 were to represent a database schema, we would have to specify two additional properties: (1) whether an employee can be neither an engineer nor a secretary, and (2) whether employees and secretaries form separate sets. We will leave these properties aside.
3. Except of course for the naming of the new concepts, which will be discussed in a further section.

Note

The *subconcept/superconcept* relation that holds among the set of concept is an important feature of all the knowledge representation systems (the conceptual schema of databases being one of them). The linguistic interpretation of this relation often takes the form *each instance of A is an instance of B*, abbreviated to *each A **is a** B*. Hence the synthetic name ***is-a***, or any variation such as ***ISA***, that is commonly used to denote this type of relation.

The solution is not always as straightforward

Sometimes, things can get more tricky. Consider for example the initial schema of Figure 30.3, which is an extension of that of Figure 30.1. While ENGINEER and SECRETARY have common attributes, it appears that ENGINEER also has some common attributes with MANAGER, but those are different from the first set: {Emp#,Name,Salary} for the first two concepts and {Emp#,Name,Level} for the last two concepts.

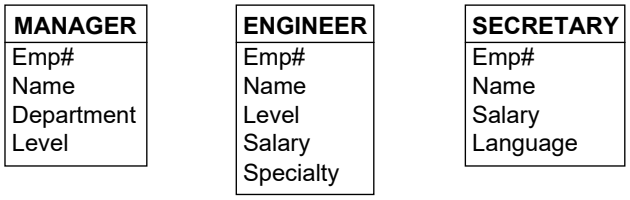

Figure 30.3 - A source schema comprising three concepts

A first attempt would consist in completing the transformed schema of Figure 30.2 with the concept MANAGER. We then observe that MANAGER and EMPLOYEE have two attributes in common, namely Emp# and Name. Applying the reasoning we used above, we create a new concept, PERSONNEL, that collects these common attributes. PERSONNEL has two subconcepts: MANAGER and EMPLOYEE, as shown in Figure 30.4.

The result is not quite satisfying. Indeed, MANAGER and ENGINEER have common attribute Level, that cannot be moved to PERSONNEL nor to EMPLOYEE and that is left unprocessed.

The correct transformation relies on the introduction of a new concept of which MANAGER and ENGINEER are the subconcepts. Let us call it OPERATIONAL and let it be assigned attribute Level. Since MANAGER also have attributes Emp# and Name, we specify OPERATIONAL as a subconcept of PERSONNEL: in this way, OPERATIONAL inherits these attributes that also are inherited by MANAGER (Figure 30.5).

This schema shows only the proper attributes of each concept. In schema 30.6, we have also introduced the inherited attributes. For each concept, it shows the proper (in blue) and inherited attributes. This view is less concise but it makes the composition of all concepts explicit.

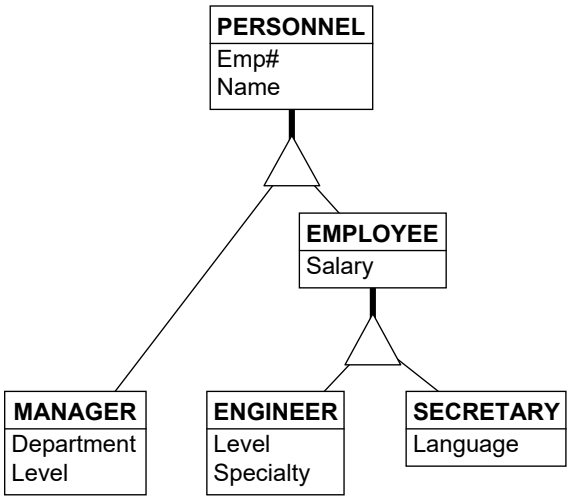

Figure 30.4 - Extracting common attributes: a first tentative

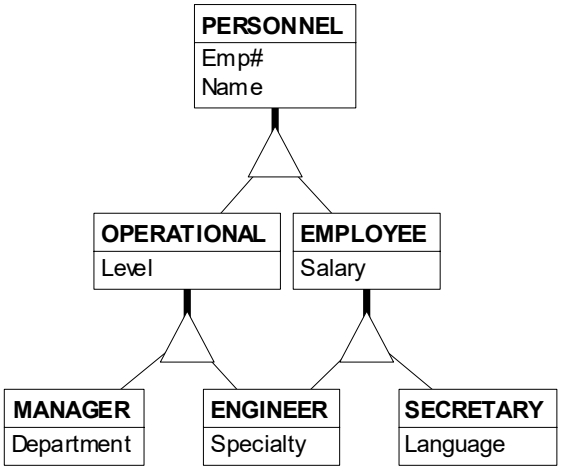

Figure 30.5 - Extracting common attributes: a more satisfying solution

Naming the concepts generated

The assignment of a semantically expressive name to each new concept is usually based on common sense and on domain knowledge. In some cases the nature of the proper attributes of a concept may suggest its name. For example, in the schema of Figure 30.5, considering that the concept EMPLOYEE represents people who receive a salary, we may have more appropriately renamed it SALARIED.

The use of AI technologies could be considered, such as domain ontologies or intelligent conversational systems such as ChatGPT or currently emerging competi-

tors (2023). However, such techniques go well beyond the scope of this (modest) study.

On this issue, it should be observed that naming new concepts is not always necessary or even useful. In the applications that illustrate FCA techniques in the scientific literature, a concept is defined entirely by a set of objects and a set of attributes. Thus, what we have called EMPLOYEE in Schema 30.5 will simply be identified as the unnamed concept

```
(ENGINEER,SECRETARY)x(Emp#,Name,Salary)
```

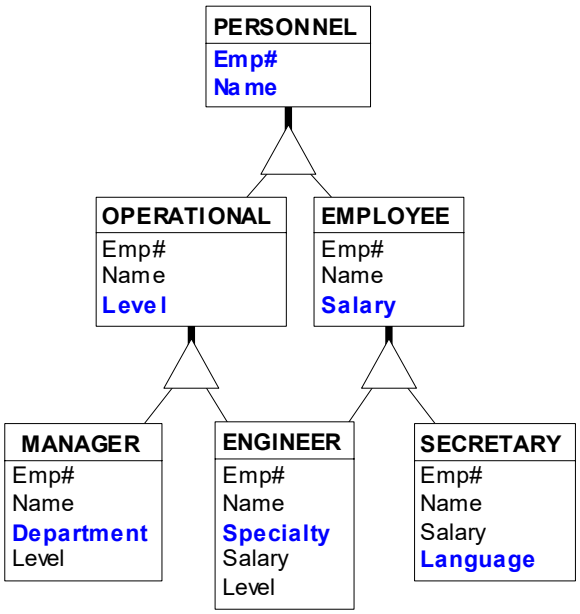

Figure 30.6 - The final schema showing both the proper and inherited attributes

Some words of conclusion

Starting from the composition (the set of their attributes) of initial concepts MANAGER, ENGINEER and SECRETARY, we have discovered, through symbolic (or *formal*) techniques, three new concepts of interest. This is the essence of the *Formal Concept Analysis* and the explanation of its name.

Though we have not distinguished them explicitly, the construction of the final schema comprises two distinct phases, that require specific reasonings and to which we will devote different algorithms:

– finding the concepts,

– building the subconcept/superconcept hierarchy.

The way we have built the final schema is fairly intuitive and obviously cannot solve more complex problems that may comprise thousands of initial objects[4]. So, we have to design a systematic procedure to discover the implicit concepts, which is our goal in the next sections.

What next?

In the next section (Section 30.2), we analyze the general principles of FCA that underlie the discovery of the concepts. In Section 30.3 we study the building of the concept hierarchy. In Section 30.4, we develop a systematic procedure to discover the concepts.

Section 30.5 describes the Chein algorithm, one of the most popular technique that translates this procedure. This algorithm is implemented in SQL in Section 30.6 and in Python in Section 30.7. The implementation in SQL and Python of the building of the concept hierarchy is described in Section 30.8. A alternative algorithm, specially designed for database schema processing, is described in Section 30.9.

Section 30.10 is devoted to the presentation of experimental material (a portfolio of typical problems and four SQL scripts to solve them) while Section 30.11 evaluates the performance of the four implementations according to qualitative and quantitative criteria. Section 30.12 suggests some ways to improve the execution times of the algorithms.

About the scope of this study

It is important to recall the scope of the series of case studies in which this one is part. Their objective is to examine and evaluate the expressive power of database concepts and technologies in modeling and solving a wide variety of problems. They do not constitute a theoretical statement or a state of the art of the problem domain they address. Based on one or more representative examples, each study attempts to derive modeling and solving principles that can help readers and practitioner feel more comfortable in dealing with such problems.

In this study, we describe a very small subset of the FCA domain that is far from encompassing all aspects of it. In particular, the mathematical basis of FCA will be limited to the part required to describe the selected problems and to develop solutions and their implementation.

4. Databases often include hundreds of tables (the figure in Figure 30.3 includes only three). The analysis of emails or tweets, tagged with keywords (their attributes), can lead to the processing of several thousands of messages.

30.2 Principles of FCA - Finding the concepts

The techniques of *Formal Concept Analysis* contribute to two objectives: finding the concepts that underlie the source data set and ordering these concepts in order to build the concept hierarchy. In this section, we explore the first objective.

In the FCA world, the initial problem is specified as a *set objects*, each object being *assigned* a *set of attributes*. In the examples examined in the previous section, these initial objects already were concepts. This is due to the particular nature of the problem domain, that of data structures. FCA considers a more general domain, in which a concept is a subset of objects that share common attributes. Some objects may become concepts while others will be absorbed by new concepts and therefore will disappear.

This initial problem is called the **Formal context**. It comprises three parts, `G`, a set of objects, `M`, a set of attributes[5] and `I(G,M)`, the attribute assignment relation. The fact `(g,m)` ∈ `I` indicates that object `g` has attribute `m`. Actually, the only interesting part of this specification is relation `I` (in this section we ignore objects without attributes and attributes that do not appear in any object). As this relation is binary, it can be represented by a matrix[6] as illustrated by Figure 30.7, that expresses the schema of Figure 30.3 (we ignore the gray cells for now). The object and attribute names have been abbreviated to fit in the limited width of this page.

Each row represents a source object and its attributes while each column represents a source attribute and all the objects it is associated with. A cross at the intersection of an object row and an attribute column (i.e., a *cell*) indicates that this object has this attribute. The matrix shows graphically that the SEC object has the attributes Emp, Nam, Sal and Lan. It also shows that the Sal attribute is assigned to the ENG and SEC objects.

	Emp	Nam	Dep	Lev	Sal	Spe	Lan
MAN	x	x	x	x			
ENG	x	x		x	x	x	
SEC	x	x			x		x

Figure 30.7 - Matrix representation of a formal context

We postulate that the order of the rows and that of the columns are irrelevant. If we swap two rows or two columns, the resulting matrix provides exactly the same information.

Relation `I` can be given various equivalent representations. With database structure in mind, we could think of a direct translation of the mathematical expression:

5. These symbols come from their German names: G = Gegenstände; M = Merkmale.
6. Called *incidence matrix* by mathematicians.

```
create table I(G varchar(256),M varchar(256));
insert into I values ('MAN','Emp'),('MAN','Nam'), ...;
```

We will sometimes use a more concise form in which we associate with each object name the **list** of the attribute names ; a little more tricky to manipulate by algorithms but better fitted to manual data entry:

```
create table I(G varchar(256),M varchar(256));
insert into I values ('MAN','Emp,Nam,Dep,Lev'), ...;
```

Rectangles

The key concept of FCA is the **rectangle**. A rectangle is made up of a set of objects **A** and a set of attributes **B** such that all the cells defined by their intersection include a cross. It will be noted **AxB**. Since the order of the rows and the columns are meaningless, the rows (and the columns) of a rectangle need not be contiguous. For example, in the matrix of Figure 30.7, `(ENG,SEC)x(Nam,Sal)` is a rectangle. `(SEC)x(Emp,Nam,Sal,Lan)` and `(MAN)x(Dep)` are also valid rectangles. Two operators will help us reason about the properties of rectangles:

- **`att(A)`**, where `A` is a set of objects, denotes the set of attributes shared by all the objects in `A`
- **`obj(B)`**, where `B` is a set of attributes, denotes the set of objects which share all the attributes in `B`.

Some examples:

```
att(MAN) = (Emp,Nam,Dep,Lev)
att(MAN,SEC) = (Emp,Nam)
obj(Nam,Sal) = (ENG,SEC)
```

Extending a rectangle

Sometimes, a rectangle can be **extended** by adding to it either an object or an attribute. For example, if we add attribute `Emp` to rectangle `(ENG,SEC)x(Nam,Sal)`, we get the new valid rectangle `(ENG,SEC)x(Emp,Nam,Sal)` shown on grey background in Figure 30.7. The condition is that the additional attribute appears in (at least) all the objects of the rectangle. Conversely, we can add an object to a rectangle if it is assigned (at least) all the attributes of this rectangle. Following these rules, we could not add object MAN (which has no Sal attribute) nor attribute Lan (which is not assigned to ENG) to this rectangle.

Maximal rectangles and concepts

Among the (many) rectangles of the matrix, the **maximal rectangles** are particularly remarkable. A rectangle is maximal if there is no way to extend it. Considering `A` a set of objects and `B` a set of attributes, rectangle `AxB` is maximal if, and only if,

`att(A) = B` and `obj(B) = A`

Explanation:

The first part tells that all the attributes of set `B` are shared by all the objects of set `A`; in other words, there is no attribute outside of `B` that would be shared by all the objects of `A`. According to the second part, there is no object outside of `A` that would be assigned all the attributes of `B`. Therefore, there is no object and no attribute that can be added to a maximal rectangle.

Now, the fundamental definition of FCA:

> *in a definite context, each **concept** is a maximal rectangle and each **maximal rectangle** is a concept.*

The intuitive idea behind this definition is that a set of objects that all share the same set of attributes (and this, exclusively), surely reveals a semantic entity hidden among all the objects of the context.

So, we are able to state our first objective, *finding the maximal rectangles in a formal context*.

Finding concepts by visual inspection

Carefully examining the matrix of Figure 30.3 to identify its maximal rectangles is a nice exercise to check if we have understood the definition of a concept. If we try, we should find this set of concepts,

```
(MAN)x(Emp,Nam,Dep,Lev)
(SEC)x(Emp,Nam,Sal,Lan)
(ENG)x(Emp,Nam,Lev,Sal,Spe)
(MAN,ENG)x(Emp,Nam,Lev)
(ENG,SEC)x(Emp,Nam,Sal)
(MAN,ENG,SEC)x(Emp,Nam)
```

of which we derive the graphical representation of figure 30.8.

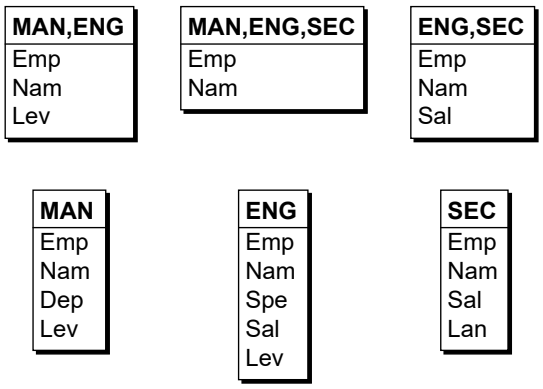

Figure 30.8 - Graphical representation of the concepts

Merging two rectangles

To formalize a procedure to find the concepts of a context, we need a new basic operator: **merging** two concepts.

If we find two rectangles A1xB1 and A2xB2 such that sets B1 and B2 have some common attributes (i.e., their intersection is not empty), then we can **merge** them to derive a new valid rectangle defined as follows:

$$(A1 \cup A2) \times (B1 \cap B2)$$

We can find several examples of this pattern in the set of concepts we have found in the previous section. Let us consider this one,

(ENG)x(Emp,Nam,Lev,Sal)
(SEC)x(Emp,Nam,Sal,Lan)

Since their attribute parts have three common attributes, we can merge them to derive this concept:

(ENG,SEC)x(Emp,Nam,Sal)

If we take a look at the matrix of Figure 30.7, we observe that this derived concept is precisely the gray rectangle.

As another example, the following concepts, that share attributes Emp and Nam,

(MAN,ENG)x(Emp,Nam,Lev)
(ENG,SEC)x(Emp,Nam,Sal)

can be merged to produce this one:

(MAN,ENG,SEC)x(Emp,Nam)

Degenerated merging

Interestingly, if $B1 = B2$, the merging operator reduces to

$$(A1 \cup A2) \times B1$$

which is the precise description of the *extension of a rectangle by adding an object*, or, more generally, *a set of objects*.

This rule can be extended to $B1 \subseteq B2$, with the same result: a new rectangle defined by $(A1 \cup A2) \times B1$. We do not know whether this new rectangle is maximal, but we know for sure that $A1 \times B1$ definitely **is not** a maximum rectangle and will not appear in the final schema.

Processing non-binary relations

The FCA is based on a binary relation, **I(G,M)**, indicating whether the object $g \in G$ is characterized by the attribute $m \in M$. For example, in a context describing the characteristics of a set of people, we distinguish those who have a university degree from those who do not. However, we would find it more interesting to also know the

faculty in which they obtained this degree. In other words, we are interested not only in the existence of an attribute but also in its value for a definite object. Ideally, we would need a ternary extension of `I` defined as `I'(G,M,V)` where `V` is a set of values. So, the context would tell us that *Smith* has no *university degree* and *Johnson* has a *university degree* in *Humanities*.

This can be done by replacing the initial single attribute by a series of attributes the names of which include a value[7]. Of course, this only works if the set of values is not too large. In the example described above, the attribute

UDegree

could be expanded into this series of attributes

UDegree:Arts, UDegree:Classics, UDegree:Humanities, *etc*.

30.3 Principles of FCA - Building the concept hierarchy

The second step of FCA aims to create the subconcept/superconcept hierarchy. We will see that it is the easy part of the FCA procedure.

If we examine very carefully the schema of figure 30.8, and if we keep in mind the concept hierarchy we built manually in figure 30.6, we can suggest a rule to automatically derive this hierarchy. Actually, two equivalent rules work equally well. Considering the pair of concepts `C1` ≡ `A1xB1` and `C2` ≡ `A2xB2`, `C1` is a subconcept of `C2` if either condition is met:

`A1` ⊂ `A2`

`B1` ⊃ `B2`

These rules rely on the assumption that the inclusion relation is asymmetric, i.e., two concepts cannot have the same object part or the same attribute part. In other words, the concepts have distinct object parts as well as distinct attribute parts.

Let us remind the concepts of figure 30.8:

```
(MAN)x(Emp,Nam,Dep,Lev)
(SEC)x(Emp,Nam,Sal,Lan)
(ENG)x(Emp,Nam,Lev,Sal,Spe)
(MAN,ENG)x(Emp,Nam,Lev)
(ENG,SEC)x(Emp,Nam,Sal)
(MAN,ENG,SEC)x(Emp,Nam)
```

From the application of the rules, we conclude that,

`(MAN)` is a subconcept of `(MAN,ENG)`

`(MAN)` is a subconcept of `(MAN,ENG,SEC)`

7. This technique is known as *One-hot Encoding* (https://en.wikipedia.org/wiki/One-hot) or *Conceptual Scaling* [Kuznetsov 2015].

(SEC) is a subconcept of (ENG,SEC)
(SEC) is a subconcept of (MAN,ENG,SEC)
(ENG) is a subconcept of (MAN,ENG)
(ENG) is a subconcept of (ENG,SEC)
(ENG) is a subconcept of (MAN,ENG,SEC)
(MAN,ENG) is a subconcept of (MAN,ENG,SEC)
(ENG,SEC) is a subconcept of (MAN,ENG,SEC)

Being transitive, the inclusion relation includes many uninteresting couples. For instance, knowing that *(MAN) is a subconcept of (MAN,ENG)* and *(MAN,ENG) is a subconcept of (MAN,ENG,SEC)*, there is no need to state that *(MAN) is a subconcept of (MAN,ENG,SEC)*. By removing these pairs, we get the *transitive reduction* of the inclusion relation, that is,

(MAN) is a subconcept of (MAN,ENG)
(SEC) is a subconcept of (ENG,SEC)
(ENG) is a subconcept of (MAN,ENG)
(ENG) is a subconcept of (ENG,SEC)
(MAN,ENG) is a subconcept of (MAN,ENG,SEC)
(ENG,SEC) is a subconcept of (MAN,ENG,SEC)

The resulting schema is shown in figure 30.9. Both their proper and inherited attributes are associated with each concept.

30.4 Looking for a systematic procedure

Let us go back to the source schema 30.3, of which Figure 30.10 is a copy (with abbreviated names). It forms a formal context comprising three objects (we do not yet consider them concepts) that can be represented by these rectangles:

```
(MAN)x(Emp,Nam,Dep,Lev)
(ENG)x(Emp,Nam,Spe,Sal,Lev)
(SEC)x(Emp,Nam,Sal,Lan)
```

30.4.1 Finding the concepts

We will apply a bottom-up procedure that starts from these three objects.

• Step 1

The source objects having different sets of attributes and none of these attribute sets being a strict subset of the others, each of these objects corresponds to a maximal rectangle and can be declared a *concept*.

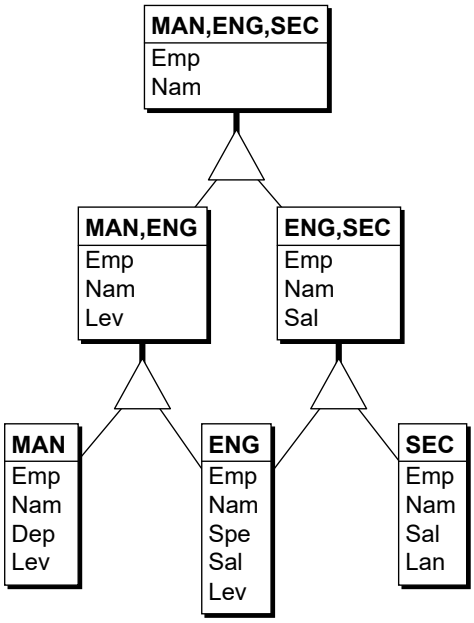

Figure 30.9 - The concept hierarchy is built from the transitive reduction of the inclusion relation of the object parts of the concepts

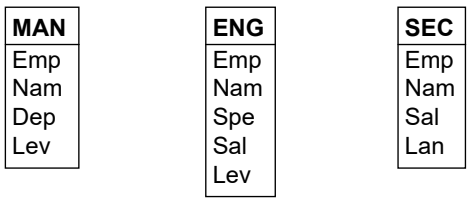

Figure 30.10 - Reminder of the source objects of figure 30.3

• Step 2

Then, we examine all the pairs of these rectangles to identify their common attributes:

– (MAN) and (ENG) : Emp, Nam, Lev

– (MAN) and (SEC) : Emp, Nam

– (ENG) and (SEC) : Emp, Nam, Sal

We create a rectangle by **merging** the rectangles of each of these pairs as follows:

– its name is the *union* of the names of the object sets of the pair

– its attributes are the *intersection* of the names of the attribute sets of the pair, provided this intersection is not empty

So, we create three new rectangles,

```
(MAN,ENG)x(Emp,Nam,Lev)
(MAN,SEC)x(Emp,Nam)
(ENG,SEC)x(Emp,Nam,Sal)
```

They are depicted in Figure 30.11.

Since the attribute parts of these rectangles are different, there is no need to merge them. Furthermore, when generating one of the new rectangles, no source rectangles had an attribute part that was identical to that of the generated rectangle (otherwise we could have discarded this source rectangle).

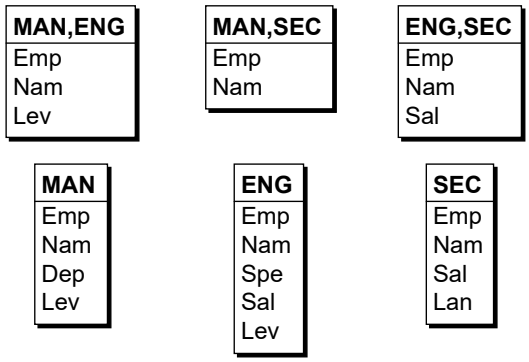

Figure 30.11 - Merging pairs of source rectangles - First level

• Step 3

So far, so good. Now, we examine each pair of the new rectangles and we consider their common attributes:

– `(MAN,ENG)` and `(MAN,SEC)`: `Emp,Nam`

– `(MAN,SEC)` and `(ENG,SEC)`: `Emp,Nam`

– `(MAN,ENG)` and `(ENG,SEC)`: `Emp,Nam`

that are depicted in Figure 30.12.

The attribute parts of each pair intersect, so that a new rectangle can be generated from this pair (Figure 30.12):

```
(MAN,ENG,SEC)x(Emp,Nam)
(MAN,SEC,ENG)x(Emp,Nam)
(MAN,ENG,SEC)x(Emp,Nam)
```

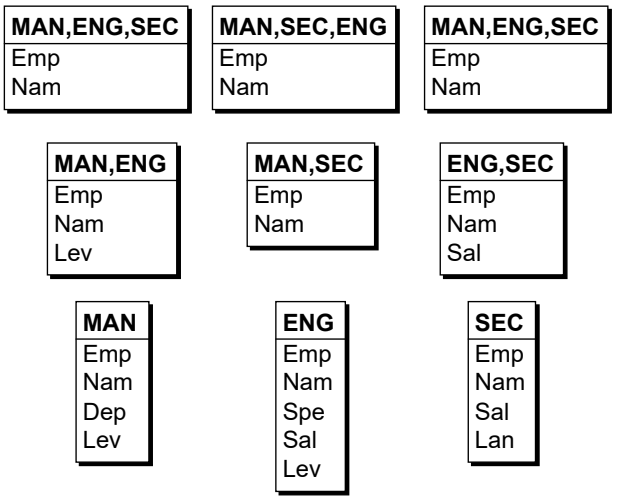

Figure 30.12 - The final schema, before cleaning

• Step 4

These three rectangles happen to be exactly the same. We only need one! We notice that the attribute part of `(MAN,SEC)`, i.e., `(Emp,Nam)`, is the same as that of the rectangles it helped generate. Therefore, this source rectangle is not maximal and we can discard it.

The top level consists of only one (maximal) rectangle, which completes the process. The final schema comprises all the rectangles that have not been discarded and that are interpreted as the six concepts of Figure 30.13.

30.4.2 Building the concept hierarchy

The objective is to build a binary relation ISA(Sub,Super) on the set of concepts such that if (C1,C2) ∈ ISA, then C1 is a subconcept of C2. We suggest a simple and intuitive technique in two steps.

In the first one, we examine each pair (C1 ≡ A1xB1, C2 ≡ A2xB2) of concepts and we check whether the object part of one of them is included in the object part of the other. If A1 ⊂ A2 then (C1,C2) ∈ ISA.

The second step discards the transitive couples from ISA, that is, it computes its *transitive reduction*. For each couple (Cs,CS) of ISA, we search ISA for any pair of couples ((C1s,C1S), (C2s,C2S)) such that

Cs = C1s and C1S = C2s and C2S = CS

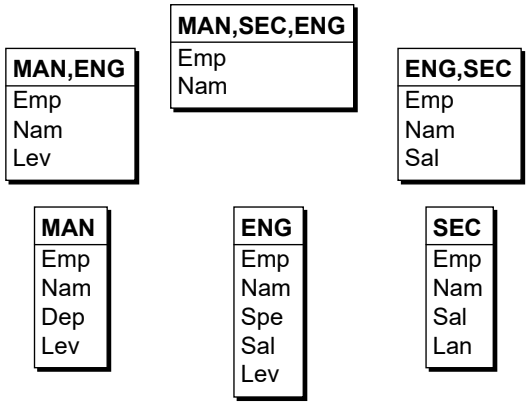

Figure 30.13 - The six concepts of the final schema

If we find such a pair of couples, then (Cs,CS) is transitive and is discarded. The ISA couples that remain form the desired transitive reduction. Deriving the subconcept/ superconcept relation of the demonstration case of Figure 30.13 is immediate (see Figure 30.9).

30.5 The *Chein* algorithm

The operations described in the preceding section can be assembled and synthesized into a formal procedure, which we will call *the Chein algorithm*. The mathematical foundation of this algorithm has been described (more precisely *sketched*) in a research note by M. Chein [Chein, 1969] and has been discussed in many further references, for example in [Sarmah, 2013].

We will develop two implementations of this algorithm. The first one expresses it into a sequence of SQL queries. The second one is a direct translation in Python.

30.5.1 Analysis of the algorithm

The manual procedure has a bottom-up layered structure that is general enough to be formalized as an iterative algorithm. Each iteration creates a new level of rectangles based on the analysis of the set of rectangles generated by the previous level.

Let us call `R1` the set of the rectangles created in the previous level and `R2` the set of rectangles being created in the current level.

The content of `R1` in the **first** level is made up of the rectangles created on the objects of the context.

When the processing of the **current** level starts, the set `R1` comprises rectangles, some of which may be maximal, and the set `R2` is empty. Then, we consider all the

pairs of distinct rectangles of R1. Let $r_i = (g_i) \times (m_i)$ and $r_j = (g_j) \times (m_j)$ be such a pair under examination. When we compare their structure, the following cases may arise:

– These rectangles share no common attributes ($m_i \cap m_j$ is empty): we abandon this pair.

– These rectangles have some common attributes, denoted by $m_{ij} = m_i \cap m_j$: there is an opportunity to merge them into a new rectangle $r_{ij} = (g_{ij}) \times (m_{ij})$, where $g_{ij} = g_i \cup g_j$. Then, we consider two situations:

  • there is no rectangle in R2 that has the same attribute part m_{ij}: we merge r_i and r_j to add the new rectangle r_{ij} to R2.

  • on the contrary, R2 happens to already include a rectangle $r_k = (g_k) \times (m_k)$, such that $m_k = m_{ij}$: we augment r_k with r_{ij}, which gives it this new composition: $r_k = (g_k \cup g_{ij}) \times (m_k)$.

When rectangle r_{ij} has been either added or merged into r_k, we compare it to its source rectangles r_i and r_j from R1. First, we observe that g_{ij} is always larger than g_i and g_j.[8] Let us suppose that $m_i = m_{ij}$. This means that the new rectangle r_{ij} extends the source rectangle r_i. Therefore, r_i is not maximal and will not be part of the solution. To state this fact, we mark r_i as *discarded*. Otherwise, if $m_i \neq m_{ij}$, r_i still is a potential maximal rectangle. At the end of the current level, all the rectangles of R1 that have not been marked as discarded definitely are maximal rectangles.

When, at the completion of a level, R2 is empty or contains one rectangle only, the analysis is completed. This last rectangle, if any, is maximal.

All the maximal rectangles identified in the successive levels of the algorithm form the concepts that can be extracted from the source context.

Before developing implementations of this algorithm, let us synthesize the main operations carried out during each level of the FCA algorithm:

Let set R, initially empty, contain the solution, that is, all the maximal rectangles.

for each pair (r_i, r_j) of distinct rectangles in R1 that match

– we merge them into rectangle $r_{ij} = (g_i \cup g_j) \times (m_i \cap m_j)$

– we search the current state of R2 for another rectangle r_k with the same attribute part as r_{ij}; if we find one we extend r_k with r_{ij}; otherwise, we add r_{ij} to R2

– if one of the arguments r_i or r_j of the merging has the same attribute part as r_{ij}, we discard it from R1

At the end of the level, all the rectangles in R1 that have not been discarded are maximal rectangles and therefore are saved in R.

In the next level, R2 replaces R1 and is emptied

8. At any time of the analysis process, all the rectangles in R1 have distinct g parts by construction. So, their union is always larger that each of them.

30.5.2 Procedural expression of the Chein algorithm

Now, the Chein algorithm can be precisely structured as an iterative procedure in which each iteration builds a level[9]. During the current iteration, that creates the current level, we consider three sets:

- `R1`: the set of rectangles built by the preceding iteration
- `R2`: the set of rectangles being built by the current iteration
- `R`: the set of maximal rectangles identified so far.

`R = { }`
Initially, `R2` contains the rectangles created from each object of the formal context and its attributes; they are not marked as discarded.
while `R2` comprises at least two rectangles:
`R1 = R2`
`R2 = { }`
for each pair (r_i,r_j) of distinct rectangles of `R1`:
$m_{ij} = m_i \cap m_j$
$g_{ij} = g_i \cup g_j$
if $m_{ij} \neq \{\}$:
if `R2` includes a rectangle $r_k = (g_k) \times (m_{ij})$:
extend r_k with g_{ij} (i.e., $r_k = (g_k \cup g_{ij}) \times (m_k)$)
else
add rectangle $(g_{ij}) \times (m_{ij})$ to `R2`
endif
if $m_i = m_{ij}$: discard r_i from `R1` **endif**
if $m_j = m_{ij}$: discard r_j from `R1` **endif**
endif
endfor
add non discarded rectangles of `R1` to `R`
endwhile
add `R2` to `R`

Script 30.1 - Pseudo-code of the Chein algorithm

9. This version of the Chein algorithm is a bit different from that that can be found in the literature, where the matching pairs are built from undiscarded rectangles of `R1`. Unfortunately, this algorithm is erroneous. A rectangle that has just been tagged as 'discarded' cannot be ignored since it may still generate other matching rectangles.

30.6 An SQL implementation of the *Chein* algorithm

Rewriting the Chein algorithm into an SQL procedure would be a very bad idea. SQL being a *set-oriented* language, a modification query (`update`, `delete`, `insert-select`) is designed to work at its best on a set of rows, not on individual rows. Such an SQL algorithm, qualified in the database jargon of *row-at-a-time*, is likely to be a variant of the Python procedure in which each access to, and modification of, `R1` and `R2` rows has been replaced by a single-row SQL query, which would be terribly inefficient!

Converting the *row-at-a-time* algorithm into a set-oriented algorithm is fairly straightforward provided we look at the problem from a slightly different angle.

Within a definite level, we will have to carry out four basic operations:

1. extracting matching pairs from `R1` to form new rectangles and storing them into `R2`
2. updating the rectangles of `R2` that have the same attribute part as some new rectangles
3. discarding the rectangles of `R1` that were one of the two sources of some new rectangle of `R2`
4. saving the undiscarded rectangles of `R1` in `R`.

The main difference between procedural and SQL algorithms is that the processing of matching pairs, which is **immediate** as soon as each of them has been discovered in the procedural algorithm, is **delayed** until all of them have been discovered in the set-oriented algorithm. Both versions yield the same result.

Before developing the script based on the set-oriented algorithm, we must discuss the best way to represent and process sets of values in SQL.

30.6.1 Operations on sets in SQL

Let us recall that a rectangle is defined as a pair of sets: a set of objects and a set of attributes. The Chein algorithm applies three operations on these sets:

- intersection (of attribute sets in R1)
- union (of object sets in R1 and R2)
- checking equality (of attribute sets in R2 or between R2 and R1)

SQL provides native set operators to derive the union, the intersection and the difference of two sets.

Let us consider that the relation `I` of the source context is implemented by the table OBJ_has_ATT(ObjID,AttID). The set of attributes of object `'o'` is expressed by:

```
select AttID from OBJ_has_ATT where ObjID = 'o'
```

and the intersection of the attribute sets of objects `'o1'` and `'o2'`:

```
select AttID from OBJ_has_ATT where ObjID = 'o1'
   intersect
select AttID from OBJ_has_ATT where ObjID = 'o2'
```

Finally, objects `'o1'` and `'o2'` can be merged if:

```
exists(select AttID from OBJ_has_ATT where ObjID = 'o1'
         intersect
       select AttID from OBJ_has_ATT where ObjID = 'o2')
```

Now, extracting the pairs of objects that can be merged is straightforward:

```
select O1.ObjID, O2.ObjID
from  OBJ_has_ATT O1, OBJ_has_ATT O2
where exists(select AttID from OBJ_has_ATT
             where  ObjID = O1.ObjID
               intersect
             select AttID from OBJ_has_ATT
             where  ObjID = O2.ObjID);
```

This query is fine to cope with individual objects but does not help clearly if we want to process rectangles: objects have a unique identifier (ObjID) but rectangles have no obvious way to uniquely identify them. For examples, these rectangles are distinct:

```
('o1','o2')*('a1','a2') and ('o1','o2')*('a1','a2','a3')
```

similarly, these ones are also distinct:

```
('o1','o2')*('a1','a2') and ('o1','o2','o3')*('a1','a2')
```

So, in general, neither the object part nor the attribute part alone can be used as rectangle identifier. We will experience similar problems with the union of sets.

Checking the equality of two sets is a bit tricky. Given sets S1 and S2, one of the most popular equality expression checks two properties: there is no element of S1 that is not in S2, and S1 and S2 have the same size. Let us check if table `OBJ_has_ATT` includes objects with the same set of attributes:

```
select distinct O1.ObjID,'equal',O2.ObjID
from   OBJ_has_ATT as O1,
       OBJ_has_ATT as O2
where  O1.ObjID < O2.ObjID
and  not exists (select *
                 from   OBJ_has_ATT
                 where  ObjID = O1.ObjID
                 and    AttID not in (select AttID
                                      from   OBJ_has_ATT
                                      where  ObjID = O2.ObjID))
and  (select count(*) from OBJ_has_ATT where ObjID = O1.ObjID)
   = (select count(*) from OBJ_has_ATT where ObjID = O2.ObjID);
```

The query examines the self-join of OBJ_has_ATT (denoted by aliases O1 and O2) and extracts its ObjID values if:

– these values are in increasing order, an optimization that ensures that (1) no object is compared to itself and (2) if two objects are equivalent, only one of them is provided,

– all the attributes of object O1 also are attributes of object O2,

– objects O1 and O2 have the same number of attributes.

This query has a theoretical complexity of $O(N^3)$,[10] where N is the size of OBJ_has_ATT, which is not particularly scalable. Executed on table OBJ_has_ATT with 660 rows (60 objects with exactly 11 attributes), this query runs in 31 s., to find the single equality. However, an index created on column ObjID allows this time to drop to 3 s, which still is very high.

Solving a basic problem: identifying a set

Basically, the problem lies in finding a convenient way to identify a set of values among a set of such sets. A particular aspect of this problem is to determine whether two sets are the same, that is, whether each element of one of them also belongs to the other one. So, we must find a unique value[11] that is representative of a set and that unambiguously denotes this set among other similar sets.

To clarify the discussion, we call *value-set* any set of values and **S** the set of *value-sets*. In addition, we admit that the values of a set can be ordered, which is the case of character, numeric and date/time values for instance. By definition, all the value-sets are distinct in **S**.

We distinguish two cases.

– The sets in **S** are disjoint. So, any element of a value-set can be used to identify it. We just need a simple rule to unambiguously select this value, for instance, the first one in ascending order within the value-set.

– The sets in **S** are not disjoint. The only reliable identifier of a value-set is its own composition or a bijective function of it, such as its secure hash[12]. To allow scalar operations on value-sets, we choose to derive a character string from the composition of each set by concatenating the ordered values of the set, separated by a character that cannot appear in the values.

  Consider for example attribute set {`Emp,Nam,Lev`} that we represent by value-set {`'Emp','Nam','Lev'`}. If no attribute name may include symbol `','`, this value-set can be represented by character string `'Emp,Nam,Lev'`. Comparing two value-sets represented in this way is not easy. For instance, one must determine that identifiers `'Emp,Nam,Lev'` and `'Nam,Emp,Lev'` denote

10. The table is joined twice with itself.
11. Also called the *signature* of the set.
12. Hashing techniques and their properties are discussed in the case study *Blockchains*.

the same set. A simple way to solve this problem is to order the values in the identifier. So, both identifiers are converted into `'Emp,Lev,Nam'` and denote the same set.

Since the sets we will compare in FCA algorithms generally are not disjoint, we will adopt the second representation technique that we will call *string-list*.

Applying the latter conventions, we translate the rectangles of Figure 30.8 into couples of set identifiers as follows:

```
('MAN','Emp,Dep,Lev,Nam')
('ENG','Emp,Lev,Nam,Sal,Spe')
('SEC','Emp,Lan,Nam,Sal')
('ENG,MAN','Emp,Lev,Nam')
('ENG,SEC','Emp,Nam,Sal')
('ENG,MAN,SEC','Emp,Nam')
```

Set operations revisited

The identifier of a set as a *string-list* explicitly provides all the content of the set, in such a way that it can be used instead of the set itself. As a consequence we can base the FCA implementation on this set representation. All we need is a collection of specific set operators applied on these representations.

SQL does not include standard string manipulation functions that would easily simulate such operators as union and intersection applied to *string-lists*[13] Fortunately, all RDBMS allow users to define custom functions, the so-called UDF (for *User-Defined Function*). SQLfast offers a set of *string-list* manipulation UDF. We will use the following (`s`, `s1`, `s2` are *string-lists*; `sep` is the separator symbol):

- `group_concat2(s,unique,o,dir,sep)`: replacement for the native elementary `group_concat` aggregate function of SQLite. Values `v` are concatenated, separated by symbol `sep`. Parameter `unique` specifies the `distinct` modifier (`0` = duplicates are preserved, `1` = they are discarded). Parameter `o` defines the order key and `dir` specifies the order direction (`1` = ascending, `2` = descending). This form is equivalent to MySQL `group_concat` function.
- `itemUnion(s1,s2,sep)`: returns a *string-list* of all the distinct values belonging to `s1` or `s2` (or both).
- `itemInter(s1,s2,sep)`: returns a *string-list* of all the distinct values belonging to `s1` and `s2`.
- `itemLen(s,sep)`: returns the number of values of `s`.

13. Some SQL engines offer column types structured as arrays of values. The technique explained here, through UDF, can be implemented in all engines.

30.6.2 SQL implementation of the *Chein* algorithm

The algorithm proceeds in seven steps, named S1 to S7, among which steps S3 to S7 create the current level and are iterated until all the concepts have been extracted.

S1: Creating the data structures

The table OBJ_has_ATT(ObjID,AttID) will receive the couples of relation I(G,M). Sets R1, R2 and R are implemented as the eponymous tables R1, R2 and R, in which each row describes a rectangle through the columns G and M that store the *object part* and the *attribute part* of rectangles, both coded in *string-list* format. The table R1 also comprises the column Disc, that indicates whether the rectangle is discarded. The table R2 includes two additional columns, Source1 and Source2, that reference, through their G part, the source rectangles in R1[14]. The last table R3 will be justified later.

```
create table OBJ_has_ATT (ObjID,AttID);
create table R1 (G,M,Disc);
create table R2 (G,M,Source1,Source2);
create table R3 (G,M,Disc);
create table R (G,M);
```

Script 30.2 - Creating the data structures (column types ignored)

S2: Initializing the process

The relation I(G,M) of the context is loaded in the table OBJ_has_ATT. Then, the query of Script 30.3 creates the initial state of the table R1 by storing the rectangles converted from OBJ_has_ATT into *string-list* format.

```
insert into R1
   select ObjID,group_concat2(AttID,1,AttID,1,',',),0
   from   OBJ_has_ATT group by ObjID;
```

Script 30.3 - Initialization: loading in R1 the rectangles of size 1 (one rectangle for each object)

S3: Searching R1 for matching pairs

This is the critical operation of the current level. For each pair (rec1,rec2) of rectangles from R1, the M parts of which have some attributes in common (i.e., rec1.M ∩ rec2.M is not empty), a new rectangle is inserted into R2. The G part of the

14. It may be surprising that we adopt the G part of rectangles to identify them, which seems to contradict the discussion on rectangle identification developed above. Actually, the Chein algorithm ensures that all the rectangles in R1 have distinct G parts.

source rectangles rec1 and rec2 are stored in columns Source1 and Source2 for further use.

The condition `rec1.G < rec2.G` ensures that each pair is examined only once.

```
insert into R2
   select itemUnion(rec1.G,rec2.G,','),
          itemInter(rec1.M,rec2.M,',') as MM,
          rec1.G,
          rec2.G
   from   R1 rec1,
          R1 rec2
   where  MM <> ''
   and    rec1.G < rec2.G;
```

Script 30.4 - Searching R1 for pairs of rectangles that match

S4: Extending rectangles in R2

Now, we analyze the rectangles of R2 to identify those that can be merged. All the rectangles that share the same M parts are replaced with a new rectangle the G part of which is the union of the G parts of these source rectangles.

Instead of updating the rectangles in R2 (deleting the source rectangles and inserting the new one), a supposedly costly operation, we store in table R3 the merged rectangles as well as the other rectangles that need not to be merged (Script 30.5). The query builds groups of rectangles with the same M value and concatenates the G parts within each group. We observe that there is no need to explicitly cope with *single* rectangles: they just form groups with one element only and are automatically copied to R3.

```
insert into R3
   select group_concat2(G,1,G,1,',') as "G",M,0
   from   R2
   group by M;
```

Script 30.5 - Merging the rectangles of R2 with the same attribute part

S5: Discarding rectangles in R1

For each rectangle g in table R2, we have memorized the G part of the source rectangles (columns Source1, Source2). If one of the latter has the same M part as g, it is marked as discarded.

```
update R1
set   Disc = 1
where (G,M) in (select Source1,M from R2
                  union
                select Source2,M from R2);
```

Script 30.6 - Marking in R1 the rectangles that can be discarded

S6: Saving the maximal rectangles

The rectangles of R1 that have not been marked are saved in table R.

```
insert into R
       select G,M
       from R1
       where Disc = 0;
```

Script 30.7 - Saving in R the maximal rectangles of R1

S7: Preparing the next iteration

The consolidated rectangles of R3 are stored in R1 then tables R2 and R3 are cleared. The iteration of the next level can then start.

```
delete from R1;
insert into R1
       select * from R3;
delete from R2;
delete from R3;
```

Script 30.8 - Preparing tables R1, R2 and R3 for the next step

The complete algorithm

Script 30.9 shows the final algorithm assembled from Steps S2 to S7[15].

15. We note that the last operation `"add R2 to R"` of the code of 30.1 has not been translated in this script. Actually, this operation is implicitly performed by the last iteration.

```
insert into R1
   select ObjID,group_concat2(AttID,1,AttID,1,',',),0
   from   OBJ_has_ATT group by ObjID;
while (True);
   extract N1 = select count(*) from R1;
      if ($N1$ = 0) exit;
   insert into R2
      select itemUnion(rec1.G,rec2.G,',',),
             itemInter(rec1.M,rec2.M,',') as MM,
             rec1.G,rec2.G
      from   R1 rec1, R1 rec2
      where  MM <> ''
      and    rec1.G < rec2.G;
   insert into R3
      select group_concat2(G,1,G,1,',') as "G",M,0
      from   R2
      group by M;
   update R1
   set Disc = 1
   where (G,M) in (select Source1,M from R2
                     union
                   select Source2,M from R2);
   insert into R select G,M from R1 where Disc = 0;
   delete from R1;
   insert into R1 select * from R3;
   delete from R2;
   delete from R3;
endwhile;
```

Script 30.9 - The final SQL algorithm

Step S7 revisited

As it is formulated, step S7 is not particularly elegant. Instead of copying the rows of R3 in table R1, an operation that may be costly, we could simply exchange their names, at no cost.

Let us call R1 and R3 two variables that contain the physical names of the tables that play the roles of tables R1 and R3 in each iteration. In the first iteration, the variable R1 denotes the table R1 and R3 the table R3. In the second iteration, the variables are swapped: the variable R1 denotes the table R3 (therefore avoiding copying the rectangles of R3 into R1) and R3 the table R1. And so on in the following iterations. This is shown in Script 30.10.

```
insert into R1
   select ObjID,group_concat2(AttID,1,AttID,1,',',),0
   from   OBJ_has_ATT group by ObjID;
set R1,R3 = R1,R3;
while (True);
   extract n1 = select count(*) from $R1$;
      if ($n1$ = 0) exit;
   insert into R2
      select itemUnion(rec1.G,rec2.G,',',),
             itemInter(rec1.M,rec2.M,',') as MM,
             rec1.G,rec2.G
      from   $R1$ rec1, $R1$ rec2
      where  MM <> ''
      and    rec1.G < rec2.G;
   insert into $R3$
      select group_concat2(G,1,G,1,',') as "G",M,0
      from   R2
      group by M;
   update $R1$
      set Disc = 1
      where (G,M) in (select Source1,M from R2
                      union
                      select Source2,M from R2);
   insert into R select G,M from $R1$ where Disc = 0;
   delete from $R1$;
   delete from R2;
   set R1,R3 = $R3$,$R1$;
endwhile;
```

Script 30.10 - Improved manipulation of tables R1 and R3 in successive iterations

It is important to note that the Chein algorithm associates with each object not only its own attributes, but also all its inherited attributes. This remark will make sense in the comparison between the different implementations of the algorithm.

30.7 A Python implementation of the *Chein* algorithm

Though the focus of this study is on the use of database concepts and tools to solve a wide variety of problems, it is interesting to compare this approach to more traditional problem solving paradigms, in particular those that rely on procedural languages. To this aim, we have written a Python procedure, FCA_Engine.py, which is a direct translation of the pseudo-code of Script 30.1[16].

We consider that the rectangles of sets R1, R2 and R can be indexed, that is, they can be accessed by their rank in their set, according to an arbitrary but deterministic order. The `for-endfor` loop is then developed into two embedded loops:

```
for j in range(1,len(R1)):
    # get rectangle rj
    for i in range(0,j):
        # get rectangle ri
        # examine and process the couple (ri,rj)
```

The main design decision is the way the sets R1, R2 and R are implemented. The first and simplest idea that comes in mind is to represent them by *lists*. R1 and R2 become lists of (g,m,d) tuples and R a list of (g,m) couples, where g is the object string-list, m the attribute string-list and d the discard indicator.

The structure of the `while-endwhile` loop can then be translated as shown in Script 30.11.

```
# list R2 is initialized with the source object:
while len(R2) > 1:
    R1 = R2
    R2 = []
    for j in range(1,len(R1)):
        (gj,mj,d) = R1[j]
        for i in range(0,j):
            (gi,mi,d) = R1[i]
            gij = union(gi,gj)
            mij = inter(mi,mj)
            if mij == '':
                continue
            gk,ik = getRectOfM(R2,mij)
            if gk is not None:
                R2[ik] = (union(gk,gij),mij,0)
            else:
                R2.append((gij,mij,0))
            if mi == mij:
                R1[i] = (gi,mi,1)
            if mj == mij:
                R1[j] = (gj,mj,1)
    addUndiscarded(R,R1)
if len(R2) > 0:
   addUndiscarded(R,R2)
```

Script 30.11 - The Chein algorithm - A Python implementation (excerpts)

This code calls these utility functions:

– `union(s1,s2)`: returns (as a string-list) the union of the elements of string-lists `s1` and `s2`.

16. Actually, FCA_Engine.py is a module that comprises a collection of different implementations of the Chein algorithm. More on this in Section 30.12 (*Improving the performance of the algorithms*).

- `inter(s1,s2)`: returns (as a string-list) the common elements of string-lists `s1` and `s2`.
- `getRectOfM(L,m)`: search list `L` for the rectangle whose attribute list is `m`; returns its rank in `L` and its object part; if none found, returns `-1` and `None`.
- `addUndiscarded(L1,L2)`: add to list `L1` the rectangles of `L2` that have not been discarded (`d == 0`).

30.8 Building the concept hierarchy

Building the concept hierarchy by extracting the inclusion relation of the G parts of the concepts has been described in Section 30.4.2. We translate the two steps of the procedure in SQL then in Python.

30.8.1 An SQL implementation

The first step builds in the table ISA the inclusion relation among the G parts of the concepts of table R. The first two columns, Sub and Super, store the G part of the subconcept and the superconcept respectively and the third column, Trans, will be explained later (Script 30.12). Considering two *distinct*[17] rows sub and sup of R, sub is a subconcept of sup if `sup.GE` includes `sub.G`, which is expressed by the UDF `itemInclude(s1,s2,sep)`, where `s1` and `s2` are string-lists and `sep` their value separator.

```
create table ISA(Sub,Super,Trans);
insert into ISA
   select sub.G,sup.G,0
   from   R sub, R sup
   where  itemInclude(sup.G,sub.G,',')
   and sub.G <> sup.G;
```

Script 30.12 - Building into the table ISA the inclusion relation among the concepts of R

Now ISA contains the *closure* of the inclusion relation. To clean it up, we need to identify and discard the transitive couples. This is the purpose of the query of Script 30.13. For each row **r** in ISA, we check whether there are two other rows whose composition is equal to **r**. If we find such rows, then **r** is transitive and should be ignored, which is specified by setting its column `Trans` to `1`. Finally, we delete the rows for which `Trans = 1`.

17. G is an implicit unique key of R (easy to prove), so, the condition `sub.G <> sup.G` ensures that equality is discarded.

```
update ISA
set    Trans = 1
where exists (select *
              from  ISA I1, ISA I2
              where I1.Sub = ISA.Sub
              and   I1.Super = I2.Sub
              and   I2.Super = ISA.Super);
delete from ISA where Trans = 1;
```

Script 30.13 - Computing the transitive reduction of the hierarchy (SQL version)

30.8.2 A Python implementation

The Python translation shown in Script 30.14 derives the concept hierarchy from the concepts whose definitions are stored in list `R` (built by Script 30.11 for example). The first part stores in list `rawInclusion` the transitive closure of the inclusion relation. The second part examines each couple `(con1,con2)` of `rawInclusion`: if `R` includes another concept `con3` such that both `(con1,con3)` and `(con3,con2)` exists in `rawInclusion`, this couple is transitive and is discarded.

```
rawInclusion = []
for con1 in R:
    for con2 in [con[0] for con in Concepts
                 if con[0] <> con1[0]]:
        if included(con1[0],con2):
            rawInclusion.append((con1[0],con2))

hierarchy = []
for (con1,con2) in rawInclusion:
    status = 'basic'
    for con3 in [con[0] for con in R
                 if con[0] <> con1 and con[0] <> con2]:
        if (con1,con3) in rawInclusion
            and (con3,con2) in rawInclusion:
            status = 'transitive'
            break
    if status == 'basic':
        hierarchy.append((con1,con2))
```

Script 30.14 - Computing the transitive reduction of the hierarchy (Python version)

30.9 An alternative FCA implementation: *ISA recovery*

The literature provides more than a dozen concept extraction techniques, among which the *Chein family*, that formalizes an intuitive manual procedure (see Section 30.4), is the most used in FCA tutorials.

We will now describe an alternative technique that has been applied successfully in the field of database engineering, in particular to *database reverse engineering*[18]. Its objective is to help recover the conceptual schema of a legacy database. Starting from the schema of an existing, sometimes several decades old, database or set of files, this technique identifies a hierarchy of entity types that represents its semantics. It provides a set of entity types (the other name of concepts in the database vocabulary) but is much simpler and faster than Chein's algorithm(s). However, it provides different results, closer to what a database developer expects from a normalized schema. Since it still need to compute the transitive reduction of its inclusion relation (Section 30.8) to extract the ISA hierarchy, it has been called *ISA recovery*.

30.9.1 The basic algorithm

Let us consider the Object/Attribute matrix of Figure 30.7 and pivot it to produce the equivalent matrix of Figure 30.14.

	MAN	**ENG**	**SEC**
Emp	x	x	x
Nam	x	x	x
Dep	x		
Lev	x	x	
Sal		x	x
Spe		x	
Lan			x

Figure 30.14 - Another presentation of the matrix of Figure 30.7

Instead of creating a rectangle from *each source object*, as we did in step S2 of Section 30.6.2, we create a rectangle from *each attribute row* of this matrix. These rectangles associate with each attribute the set of objects in which it appears. This operation materializes the **obj(B)** operator described in Section 30.2. Then we merge the rectangles that share the same set of objects, e.g., those built from Emp and Nam. By construction, these rectangles are maximal and therefore form a set of concepts of the source context (Figure 30.15).

18. See [Hainaut, 2002] for example.

```
(MAN,ENG,SEC)x(Emp,Nam)
(MAN)x(Dep)
(MAN,ENG)x(Lev)
(ENG,SEC)x(Sal)
(ENG)x(Spe)
(SEC)x(Lan)
```

Figure 30.15 - The concepts extracted from the context of Figure 30.14

We build the concept hierarchy as described in Section 30.4. The result is shown in Figures 30.16 and 30.17.

```
(MAN)x(Dep)       subtype of  (MAN,ENG)x(Lev)
(ENG)x(Spe)       subtype of  (MAN,ENG)x(Lev)
(ENG)x(Spe)       subtype of  (ENG,SEC)x(Sal)
(SEC)x(Lan)       subtype of  (ENG,SEC)x(Sal)
(MAN,ENG)x(Lev)   subtype of  (MAN,ENG,SEC)x(Emp,Nam)
(ENG,SEC)x(Sal)   subtype of  (MAN,ENG,SEC)x(Emp,Nam)
```

Figure 30.16 - Transitive reduction of the subtype/supertype relation

It is important to note that the procedure associates with each object only its proper attributes.

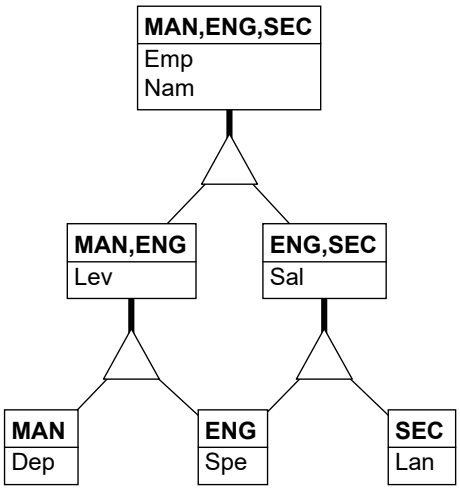

Figure 30.17 - The concept hierarchy derived from the context of Figure 30.7

30.9.2 An SQL translation of the *ISA recovery* algorithm

To implement this technique in SQL, we first define the appropriate data structures (Script 30.15).

```
create table OBJ_has_ATT(ObjID,AttID);
create table CONCEPT(G,M);
```

Script 30.15 - *ISA recovery*: the data structures (column types ignored)

Then, we extract the maximal rectangles in two steps: for each attribute, we compute the list of its objects (the `from` subquery in Script 30.16)[19], thus building elementary rectangles, then we merge the AttID parts of those of these rectangles that have the same G part. These maximal rectangles are stored in the CONCEPT table.

This procedure, particularly simple and intuitive, will be discussed and compared with the *Chein* algorithm in a later section.

30.9.3 The case of empty concepts

The basic algorithm may, in certain circumstances, create or discard concepts that have no proper attributes. We will discuss two different cases.

```
insert into CONCEPT
   select G,group_concat(AttID)
   from (select group_concat2(ObjID,1,ObjID,0,',') as G, AttID
         from   OBJ_has_ATT
         group by AttID)
   group by G;
```

Script 30.16 - Compute the concepts from the maximal rectangles in OBJ_has_ATT

Preserving the source objects with no proper attributes

If we examine in more detail the composition of the matrix of Figure 30.14, we observe that it assigns each source object (`MAN`, `ENG`, `SEC`) at least one proper attribute. The consequence is that all these source objects appear in the final set of concepts and, more explicitly, in the hierarchy of Figure 30.17. Let us suppose that we remove the attribute `Spe`, so that the attribute set of `ENG` is reduced to (`Emp,Nam,Lev,Sal`).

Applying the procedure of Script 30.16, we observe that `ENG` has disappeared. It is included in higher level aggregated concepts but no longer exists as a standalone concept:

```
(MAN,ENG,SEC)x(Emp,Nam)
(MAN)x(Dep)
(MAN,ENG)x(Lev)
(ENG,SEC)x(Sal)
```

19. We have used the extended version of `group_concat` to ensure that the components of the object lists are sorted, so that they can be *`grouped by`* correctly in the outermost query.

```
(SEC)x(Lan)
```

The new concept hierarchy is shown in Figure 30.18.

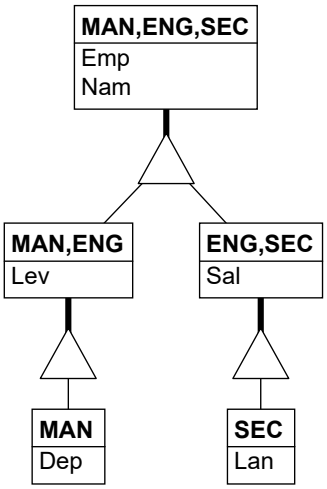

Figure 30.18 - Devoid of any proper attributes, the source object ENG disappears

Whether or not the disappearance of such source objects is desirable depends on the objective of the FCA. In the database domain, source objects most often are meaningful data sets, such as tables, object classes, table types or entity types, that are to be used in application programs. The FCA process consists in normalizing a source schema by eliciting hidden pertinent data sets without getting rid of the source objects, considered the very first concepts of the solution.

To preserve these *empty* concepts, we just add them to the CONCEPT table, as shown in Script 30.17.

```
insert into CONCEPT
   select G,group_concat(AttID)
   from (select group_concat2(ObjID,1,ObjID,0,',') as G, AttID
         from   OBJ_has_ATT
         group by AttID)
   group by G
      union
   select distinct ObjID,''
   from   OBJ_has_ATT
   where  ObjID not in (select G from CONCEPT);
```

Script 30.17 - Extended concept extraction: all the source concepts are preserved

The concept hierarchy now includes these *empty* concepts (Figures 30.19 and 30.20). It must be noted that the Chein algorithm also discards these source empty concepts.

```
(MAN,ENG,SEC)x(Emp,Nam)
(MAN,ENG)x(Lev)
(ENG,SEC)x(Sal)
(MAN)x(Dep)
(ENG)x()
(SEC)x(Lan)
```

Figure 30.19 - Reintroduction of empty source concepts

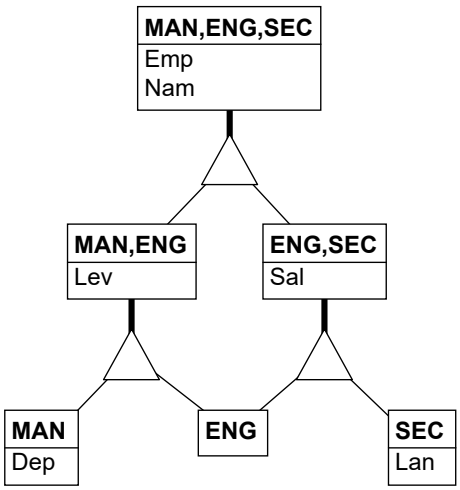

Figure 30.20 - Empty source concepts can also be represented in the concept hierarchy

Empty intermediate concepts

We consider the formal context comprising objects `MAN`, `ENG` and `SEC`, from which we have built the concept hierarchy of 30.17. Now, we add a new object to this context, ASSIST, that represents *technical assistants*, described by four attributes: `Emp`, `Nam`, `Lev` and `Sal`. The modified concept hierarchy is shown in Figure 30.21.

This hierarchy exhibits a pattern that we have not yet encountered: two concepts, `ENG` and `ASSIST`, have the same two superconcepts, `MAN,ENG,ASSIST` and `ENG,ASSIST,SEC`. More generally, *two or more concepts have at least two direct superconcepts in common*.[20]

This pattern suggests the existence of an intermediate concept, here `ENG,ASSIST`, which depends on the two superconcepts and on which the two subconcepts depend. In a sense, this new concept tells that `ENG` and `ASSIST` have a common characteristic, that is, to depend on the same set of superconcepts. It *materializes* the common

20. Or, equivalently, *a set of at least two superconcepts share the same set of at least two direct subconcepts*.

subconcept-superconcept relationship. Naturally, this concept has no proper attribute. This transformation is illustrated in Figures 30.22 and 30.23.

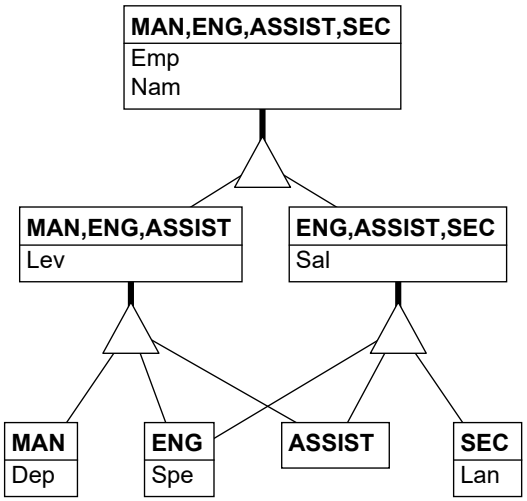

Figure 30.21 - ENG and ASSIST share the same set of superconcepts

```
(MAN,ENG,ASSIST,SEC)x(Emp,Nam)
(MAN,ENG,ASSIST)x(Lev)
(ENG,ASSIST,SEC)x(Sal)
(ENG,ASSIST)x()
(MAN)x(Dep)
(ENG)x(Spe)
(ASSIST)x()
(SEC)x(Lan)
```

Figure 30.22 - Introduction of intermediate concept ENG,ASSIST

It must be noted that the Chein algorithm automatically creates the empty intermediate concepts, whenever the pattern condition is met.

Here again, whether this transformation has to be performed depends on the application domain. In a database schema, where developers tend to avoid too deep inheritance hierarchies, considered less readable, the solution of Figure 30.21 will probably be preferred. On the contrary, when extracting the most pertinent concepts from a large data set, the users might appreciate the elicitation of such concepts as ENG,ASSIST in Figure 30.23.

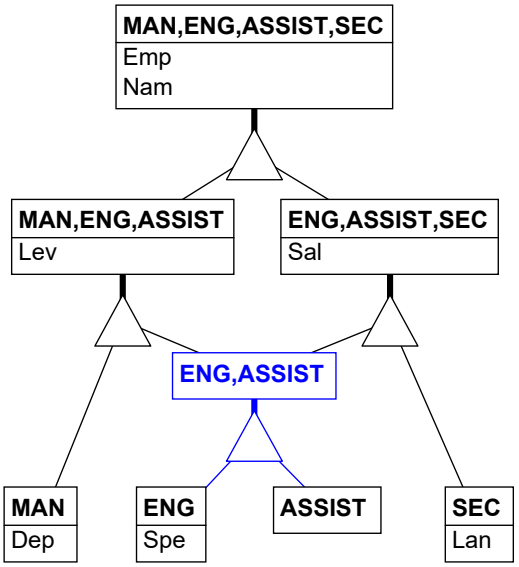

Figure 30.23 - A new empty concept makes explicit the common set of superconcepts of ENG and ASSIST

Creating the empty intermediate concepts

Let us suppose that we want to identify and create these intermediate concepts. To develop such a procedure, we will illustrate its steps by the abstract example of Figure 30.24, in which all the concepts comprise (at least) an attribute. This hierarchy has been built by the procedure developed in Section 30.8.

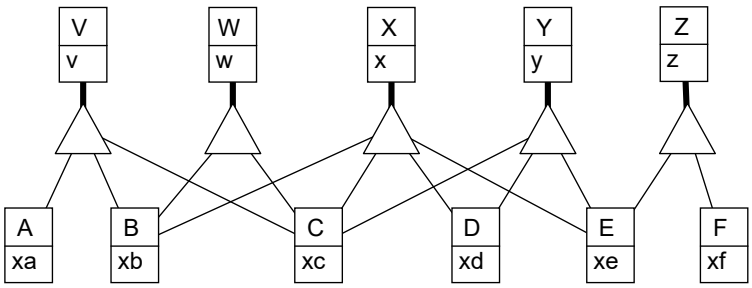

Figure 30.24 - An abstract hierarchy without empty intermediate concepts

Its fourteen *subconcept-superconcept* links are initially stored in table ISA(Sub,Super)[21]. The content of ISA is shown in Figure 30.25. It is produced by the query 30.18[22]:

21. We ignore the column Trans, now useless.

```
select Sub,group_concat(Super,';') as Supers
from   ISA
group by Sub;
```

Script 30.18 - Showing the content of table ISA

```
+-----+---------+
| Sub | Supers  |
+-----+---------+
| A   | V       |
| B   | V;W;X   |
| C   | V;W;X;Y |
| D   | X;Y     |
| E   | X;Y;Z   |
| F   | Z       |
+-----+---------+
```

Figure 30.25 - Synthetic view of table ISA

Now, we select the concepts (column Sub) that depend on *at least two superconcepts* (column Super). Concepts A and F are discarded (Script 30.19 and Figure 30.26).

```
create view ISA2(Sub,Super)
as select Sub, group_concat(Super,';')
   from   ISA
   group by Sub
   having count(*) >= 2;
```

Script 30.19 - Selecting the concepts depending on 2+ superconcepts

```
+-----+---------+
| Sub | Super   |
+-----+---------+
| B   | V;W;X   |
| C   | V;W;X;Y |
| D   | X;Y     |
| E   | X;Y;Z   |
+-----+---------+
```

Figure 30.26 - The four concepts that depend on 2+ superconcepts (view ISA2)

We compute in ISA2 the intersection of each pair of sets of superconcepts. If this intersection comprises at least two superconcepts, the subconcepts are concatenated, otherwise, the pair is dropped (Script 30.20 and Figure 30.27).

22. The choice of the `';'` separator preserves the identity of superconcepts when their name is made up of more than one component. On the contrary, subconcepts are merged (without duplicates), which justify the `','` separator, the same as that used in the concept names.

```
create view PAIR(Subs,Supers)
as select I1.Sub||','||I2.Sub,
          itemInter(I1.Super,I2.Super,';') as Supers
   from   ISA2 I1,ISA2 I2
   where  I1.Sub < I2.Sub
   and    itemLen(Supers,';') >= 2;
```

Script 30.20 - Merging pairs of concepts that share at least two superconcepts

```
+------+--------+
| Subs | Supers |
+------+--------+
| B,C  | X;W;V  |
| C,D  | Y;X    |
| C,E  | Y;X    |
| D,E  | Y;X    |
+------+--------+
```

Figure 30.27 - Pairs of concepts sharing a set of at least two superconcepts (view PAIR)

Finally, we merge the sets of subconcepts that share the same set of superconcepts (Script 30.20 and Figure 30.27). For instance, three concept sets `C,D`, `C,E` and `D,E` are merged into `C,D,E` (Figure 30.28). The updated concept hierarchy is shown in Figure 30.29.

```
select itemSort(group_concat(Subs,','),0,1,',') as NewConcept,
       Supers as SuperConcepts
from  PAIR
group by Supers;
```

Script 30.21 - Final step: merging the set of subconcepts sharing the same set of superconcepts

```
+------------+---------------+
| NewConcept | SuperConcepts |
+------------+---------------+
| B,C        | X;W;V         |
| C,D,E      | Y;X           |
+------------+---------------+
```

Figure 30.28 - The two new intermediate concepts and their superconcepts

These script fragments can be wrapped into a single query. This is left as an exercise.

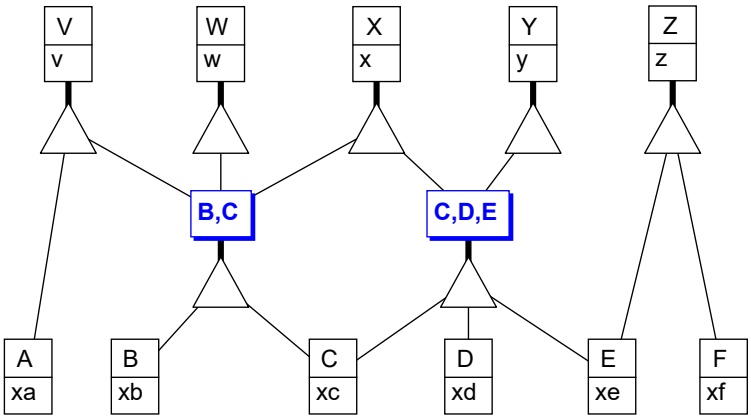

Figure 30.29 - Introducing two intermediate empty concepts

Note

The introduction of intermediate concepts can again create the conditions for the generation of additional intermediate concepts. It is therefore recommended to iterate the process until no additional concepts are created.

30.9.4 Python translation of the *ISA recovery* algorithm

Due to its simplicity, the SQL script 30.17 is easily translated into a small Python program like the one shown in Script 30.22.

```
def groupConcat(T,A,G):
    UG = list(set([t[G] for t in T]))
    AG = []
    for g in UG:
        ag = ','.join([t[A] for t in T if t[G] == g])
        AG.append((g,ag))
    return AG

innerQuery  = groupConcat(I,0,1)
Concepts = groupConcat(innerQuery,0,1)

actObj = [obj for (obj,att) in Concepts
          if len(obj.split(',')) == 1]
for obj in G:
    if obj not in actObj:
        Concepts.append((obj,''))
```

Script 30.22 - A Python translation of SQL Script 30.17

The component `I` of the context is represented by an eponymous list of couples, the first element of which is the name of an object and the second is the name of one of its attributes.

The `groupConcat` function simulates the `group_concat-group by` SQL pattern applied to a list of couples. `T` is this list, `A` is the index [0,1] of the component of the couples to aggregate and `G` is the index [0,1] of the component on which the aggregation is applied (the grouping criterion). The list `UG` comprises the unique values of the `G` components of the couples.

The first application of the function corresponds to the inner query of the SQL expression. It computes the rectangles based on each attributes. So, we aggregate the objects names (index `A = 0`) with the same attribute (index `G = 1`):

```
innerQuery = groupConcat(I,0,1)
```

The result, `innerQuery`, is a list of couples (*attribute*, *object list*). The second application merges the rectangles with the same object list. It aggregates the attribute names (index `A = 0`) with the same object list (index `G = 1`):

```
Concepts = groupConcat(innerQuery,0,1)
```

The result is that of Figure 30.15. The end of the code converts the objects with no proper attributes into concepts.

30.10 Experimentation

The algorithms developed in this study have been translated into four SQLfast scripts. They are included in the SQLfast distribution, along with a collection of sample formal contexts.

30.10.1 The scripts

Chein algorithm, Python implementation

The script **computeFCA (Python).sql** opens the control panel shown in Figure 30.30 and through which the user selects a Python procedure and the way the concept hierarchy is derived (*None*, through a *Python* algorithm, through an *SQL* script).

The Python procedure described in Section 30.7 is labelled "**1. two lists**". The other four procedures are based on alternative representations of the sets R1, R2 and R. They will be discussed in Section 30.12, devoted to performance optimization.

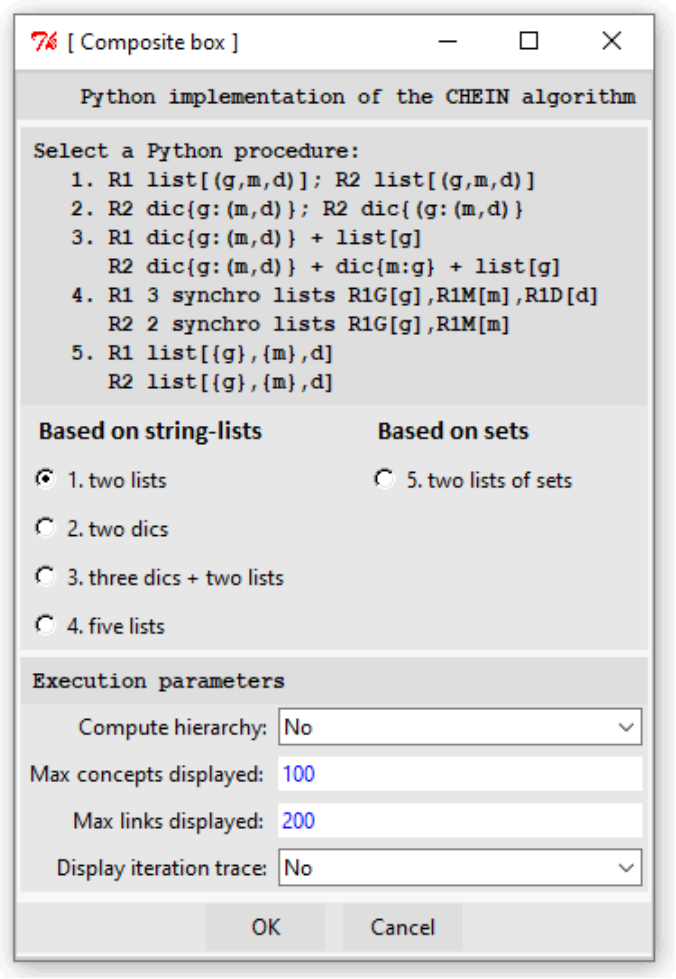

Figure 30.30 - Control panel of the Python implementation of the Chein algorithm

The script calls the **FCA_Engine.py** Python module which includes the five variants of the Chein algorithm and (optionally) the derivation of the concept hierarchy. This module has been added to the SQLfast directory.

Chein algorithm, SQL implementation

The script **computeFCA (SQL).sql** opens the control panel shown in Figure 30.31. It allows the user to specify the information to be displayed at each stage of execution and the final results. In addition, the user can evaluate the effect of various combinations of indexes on the execution time (more on this in Section 30.12.2).

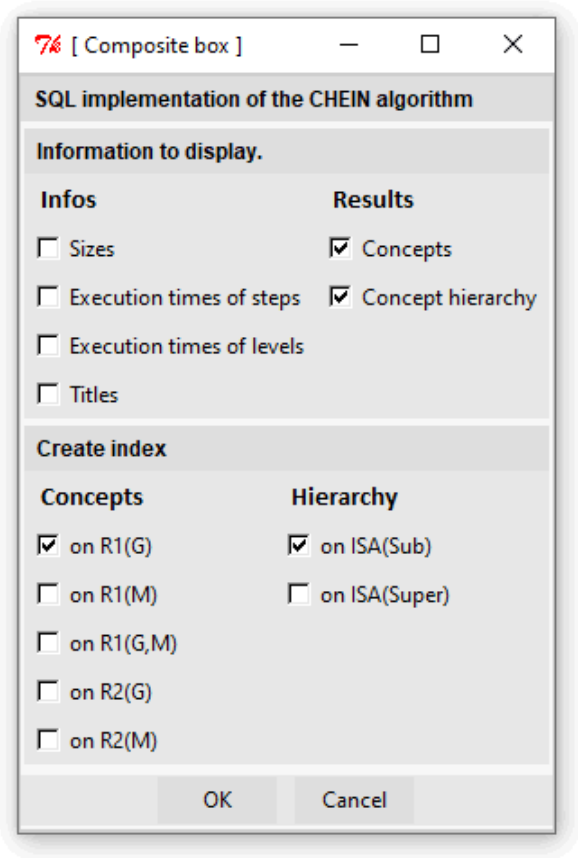

Figure 30.31 - Control panel of the SQL implementation of the Chein algorithm

ISA recovery algorithm, SQL implementation

The control panel of the script **computeISA Recovery (SQL).sql** is shown in Figure 30.32. The first frame specifies the additional concepts to include in the solution (empty source objects and empty intermediate concepts). The second frame controls the number of rows to be displayed and the maximum number of iterations when computing the intermediate concepts.

ISA recovery algorithm, Python implementation

The Python script (one of the function of the **FCA_Engine.py** module) is executed from the SQLfast script **computeISA Recovery (Python).sql**.

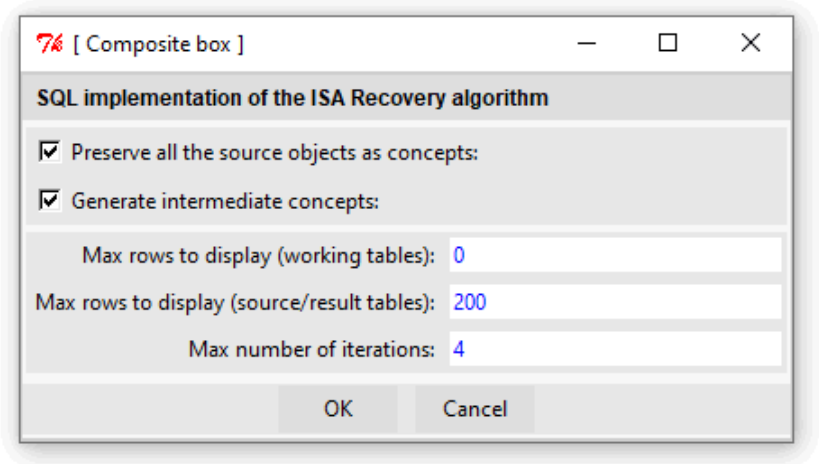

Figure 30.32 - Control panel of the ISA Recovery algorithm

30.10.2 Sample formal contexts

The literature and the tutorials available on the web provide us with many illustrative samples. However they all are quite small, comprising one or two dozen objects and not more attributes. They are fitted to play with the concepts and to illustrate the way the algorithms work, but they are less useful to compare the solution, the time performance and the space requirements of the algorithms. Some of them are included in the SQLfast distribution. They are available in the directory Scripts/Case-Studies/Case_FCA/Contexts.

– PERSONNEL

One of the first contexts used in this study.

Object	Attributes
ENG	Emp#,Name,Lev,Gra,Sal,Spe
MAN	Emp#,Name,Lev,Gra,Dep
SEC	Emp#,Name,Sal,Lang1,Lang2
TRA	Emp#,Name,Sal,Lang1

Scripts: Create-PERSONNEL.sql, FCA-PERSONNEL.csv

– **ANIMAL**

Seven animal species and their properties.[23]

Object	Attributes
canary	eggs,feather,fly,breath
crocodile	eggs,teeth,swim,breath
duck	eggs,feather,fly,swim,breath
frog	eggs,swim,breath
ostrich	eggs,feather,breath
salmon	eggs,swim
shark	eggs,teeth,swim

Scripts: Create-ANIMAL.sql, FCA-ANIMAL.csv

– **WATERS**

Description of 17 bodies of water[24].

Object	Attributes
CANAL	con,run
CHANNEL	con,run
LAGOON	con,mar,nat,sta
LAKE	con,nat,sta
MAAR	con,nat,sta
POND	con,nat,sta
POOL	con,nat,sta
PUDDLE	nat,sta,tmp
RESERVOIR	con,sta
RIVER	con,nat,run
RIVULET	con,nat,run
RUNNEL	con,nat,run
SEA	con,mar,nat,sta
STREAM	con,nat,run
TARN	con,nat,sta
TORRENT	con,nat,run
TRICKLE	con,nat,run

Scripts: Create-WATERS.sql, FCA-WATERS.csv

– **ERA**

Abstract physical relational schema (just tables and columns) derived from the flattening of an Entity-relationship conceptual schema. Comprises 25 objects and 52 attributes. *Question*: can the FCA algorithms recover the exact source conceptual schema?

23. Example often used in tutorials. See for example, https://ijcai-15.org/downloads/tutorials/T23-FCA.pdf

24. See extended description in https://en.wikipedia.org/wiki/Formal_concept_analysis

Object	Attributes
Q01	a1,a2,a3,d1,d2,j1,q01
Q02	a1,a2,a3,d1,d2,e1,j1,k1,k2,q02
Q03	a1,a2,a3,d1,d2,e1,k1,k2,q03
Q04	a1,a2,a3,b1,b2,d1,d2,e1,k1,k2,f1,f2,l1,q04
Q05	a1,a2,a3,b1,b2,e1,f1,f2,l1,q05
...	...
R06	a1,a2,a3,b1,b2,c1,c2,f1,f2,g1,g2,h1,h2,h3,m1,m2,r06
R07	b1,b2,c1,c2,h1,h2,h3,i1,i2,i3,o1,r07
R08	c1,c2,i1,i2,i3,p1,p2
R09	c1,c2,i1,i2,i3,p1,p2,r09
R10	c1,c2,i1,i2,i3,p1,p2,r10

Scripts: Create-ERA.sql, FCA-ERA.csv

– COUNTRY[25]

The formal context comprises 147 objects (countries) and 33 attributes (country indicators). Each object describes a country through 11 attributes. From this source, we have derived subsets of 10 to 147 countries. Below, the context of COUNTRY-10 (country names and attributes names are denoted by letters or numbers)

Object	Attributes
A	2,5,6,9,12,14,19,22,23,29,32
B	0,3,7,10,11,16,18,20,25,28,30
C	2,4,7,9,13,15,18,21,26,28,31
D	2,4,6,9,12,14,19,22,23,29,32
E	1,4,7,9,12,15,17,20,26,27,31
F	1,3,7,9,11,16,18,20,26,27,30
G	2,5,7,9,13,16,17,20,26,27,32
H	0,3,7,9,13,16,17,20,26,27,30
I	1,5,7,10,12,14,18,21,25,27,32
J	2,4,6,9,12,15,18,20,25,27,31

Scripts: Create-COUNTRY-10.sql (10 countries) to Create-COUNTRY-Full.sql (147 countries, FCA-COUNTRY-10.csv to FCA-COUNTRY-100.csv.

– Hypercard

This example is a reduced version of a real project the objective of which was to convert the data of a small Hypercard[26] application into a standard database. The

25. Derived from the data provided in https://github.com/mdaquin/fca.js, referenced by [D'Aquin, 2020]. The full source data are available in script Create-COUNTRY-Full.sql. Countries are coded as numbers from 0 to 146 (except in COUNTRY-10, where they are coded as letters) and attributes from 0 to 32. The interpretation of these codes can be found as comments in the full script.

very first step when converting a database is to extract its conceptual schema, in the form of a hierarchy of entity types. The application comprises eight files, the structure of which can be expressed as the following formal context:

Object	Attributes
BOO	Aut,Dat,ID,Key,Mon,Pub,Tit,Yea
COL	Dat,Edi,ID,Key,Mon,Pub,Tit,Yea
COLP	Aut,Dat,ID,Key,Pag,Sou,Tit
JOU	Dat,Edi,ID,Key,Mon,Num,Tit,Vol,Yea
JOUP	Aut,Dat,ID,Key,Pag,Sou,Tit
PRO	CDa,CTi,Dat,Edi,ID,Key,Mon,Pub,Tit,Yea
PROP	Aut,Dat,ID,Key,Pag,Sou,Tit
REP	Aut,Dat,ID,Key,Mon,Ori,Tit,Typ,Yea

The meaning of these abbreviations is described in the SQL code of the context.

Scripts: Create-HYPERCARD.sql, FCA-HYPERCARD.csv

30.10.3 The context generator

The context generator (Context-Generator.sql) is a small application that generates synthetic (i.e., artificial) formal contexts of up to 2,600 objects[27]. The attributes of each object are randomly selected among a set of attributes of up to the same size. The size of the attribute set of each object is a random number in a definite range.

The dialog box of Figure 30.33 sets the parameters for a formal context of 100 objects and 30 attributes. A set of 2 to 10 attributes will be associated with each object. The size and content of the attribute set of each object are randomly selected.

The description of the context is stored as an *SQL* script or as a *csv* file.

Due to the twofold random nature of the `I` relation, the contexts generated from the same set of parameters can vary, producing, for example, a different number of concepts.

30.11 Evaluation of the FCA implementations

In this section, we will comment and compare the four implementations developed here above on the basis of qualitative and quantitative criteria:

26. Hypercard is a user-friendly software tool (*"programming for the rest of us"*) that was very popular on the Macintosh in the 80's and beyond. Very powerful and remarkably simple, it allowed to build small interactive database applications in an intuitive way. [https://en.wikipedia.org/wiki/HyperCard]

27. The script includes a parameter to set the maximum to 26,000 or 260,000.

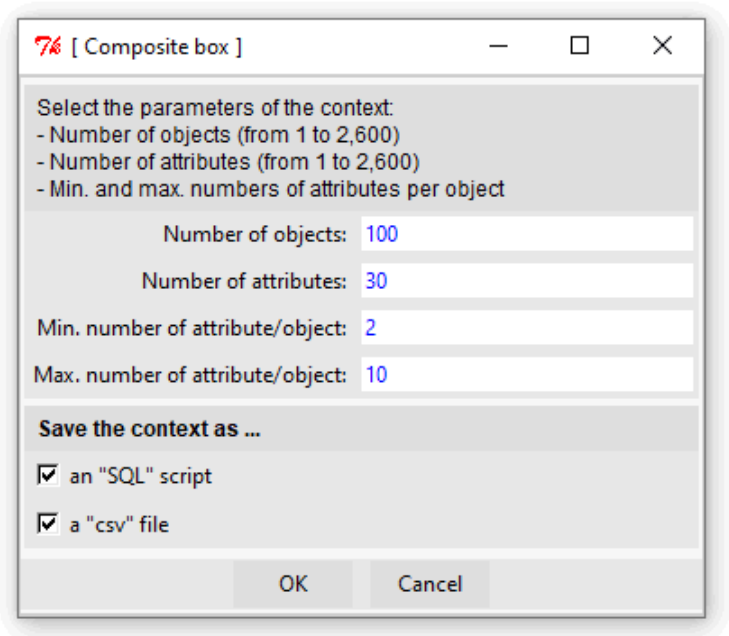

Figure 30.33 - Creating a new formal context

- Python implementation of the *Chein* algorithm
- SQL implementation of the *Chein* algorithm
- SQL implementation of the *ISA recovery* algorithm
- Python implementation of the *ISA recovery* algorithm

It should be clear that different input data sets may produce different results, at least for some metrics[28]. It would therefore be imprudent to generalize the conclusions of this section to all possible uses of the techniques studied. As usual, the reasoning is more important than the conclusions!

30.11.1 Simplicity of algorithms

Of the four FCA implementations, the SQL version of the *ISA recovery* algorithm (Script 30.17), expressed as a single intuitive query, is arguably the simplest and most natural, though some Python programmers may consider its Python translation to be equally so. The comparison between the Python and the SQL implementation of the *Chein* algorithm is a matter of taste: the Python code is shorter but the SQL

28. For example, the simplicity and source object preservation criteria do not depend on the nature of the formal context.

version which performs global and declarative set-oriented operations, may be considered more natural by some.

The comparison can change considerably when we look at *optimized versions* of the algorithms (see Section 30.12). The main technique for optimizing SQL queries does not change the queries themselves but merely adds indexes, which are external constructs that do not require table modification either. So there is no difference between the tables and the queries whether the algorithm is optimized or not. The optimized SQL solution therefore preserves its original simplicity. In contrast, optimizing a procedural program often requires changes to the data structures and algorithmic control structures such that the optimized version can be considered more convoluted and less readable.

30.11.2 Preservation of the source concepts

The *Chein* algorithm may ignore some source objects for two reasons. First, it merges the source objects with the same attribute set (step S2). Then, it discards the rectangles of R1 that have the same attribute set as some rectangles in R2 (step S5).

On the contrary, the *ISA recovery* algorithm preserves all the source objects by converting them into concepts, even those that have no proper attributes.

30.11.3 Nature of the attributes of the concepts

The *Chein* algorithm associates with each concept all its attributes, be they proper or inherited. On the contrary, the *ISA recovery* algorithm keeps their proper attributes only. When there is none, empty concepts are generated. As we have observed, they all translate source objects of the context. Unless the complementary procedure of Section 30.9.3 have been applied, each generated concept has at least one proper attribute.

Each form can be converted into the other one, either by recursively propagating the attributes of each concept to its subconcepts or by recursively deleting in each concept the attributes of its super-concepts. This is left as an exercise.

30.11.4 Creation of empty intermediate concepts

The *Chein* algorithm, through the merging step of R1 rectangles, creates new rectangles, then converts those that are maximal into concepts. Among them many intermediate concepts have no proper attributes.

The first step of the *ISA recovery* algorithm creates concepts that have proper attributes. The second step adds the concepts derived from the source objects that have no proper attributes and therefore appear empty. No empty intermediate concepts are generated. However, the complementary procedure described in Section 30.9.3 generates some empty intermediate concepts that may contribute to the readability of the final hierarchy (compare the schemas of Figures 30.24 and 30.29).

The table of Figure 30.34 shows that the number of empty intermediate concepts may be quite huge. For example, the *ISA recovery* solution of the COUTRY-50 case (labelled ISA recovery) includes 82 concepts compared to 1,639 for the *Chein* algorithm, most of them being empty intermediate concepts. Clearly, the *ISA recovery* algorithm generates less concepts than the *Chein* algorithm.

	Objects	Chein algorithm	ISA recovery
COUNTRY-10	*10*	*68*	*36*
COUNTRY-20	*20*	*299*	*50*
COUNTRY-30	*30*	*708*	*60*
COUNTRY-40	*40*	*1,187*	*72*
COUNTRY-50	*50*	*1,639*	*82*
COUNTRY-60	*60*	*2,171*	*92*

Figure 30.34 - Number of concepts generated by the algorithms. The additional empty intermediate concepts of the *ISA recovery* algorithm have not been accounted for

The figure 30.35 illustrates the same fragment of the hierarchy built from the concepts generated by both techniques.

It should be noted that each of these algorithms may generate intermediate concepts that the other does not. For example, the *ISA recovery* algorithm (through its complementary procedure), applied to the COUNTRY-10 context, generates 4 new empty intermediate concepts[29] but two of them are ignored by the *Chein* algorithm[30].

In a context of database reverse engineering, in which the physical schema of a legacy database comprises 50 tables or data files, generating a conceptual schema made up of more than 1,600 entity types has no sense[31].

In contrast, the *ISA recovery* algorithm is not suitable for discovering, analyzing and querying semantically (potentially) relevant object classes among large sets of objects.

This discussion clearly delineates the application domains of these algorithms.

30.11.5 Memory space requirements

We examine the quantity of memory space the implementation of each algorithm requires at runtime. The measurements are performed on the COUNTRY-50 context resolution.

29. CG, GH, BCI and EGHJ
30. BCI and EGHJ
31. All the more so as each concept will be, in a second phase of the database reengineering process, translated into a table!

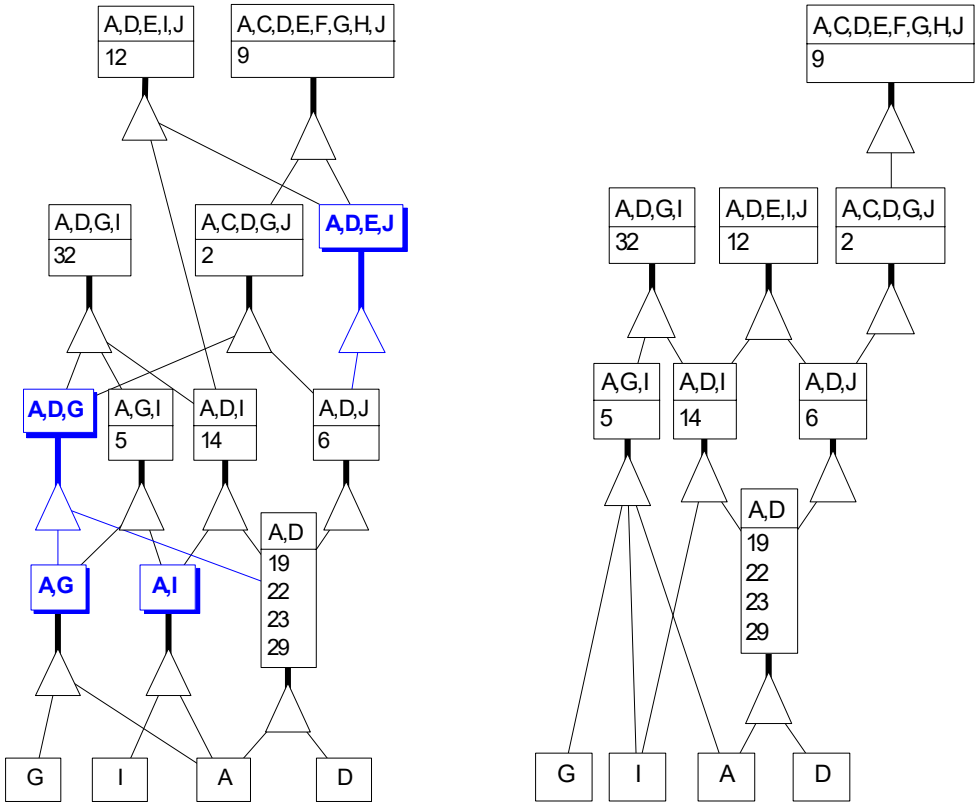

Figure 30.35 - Fragments of the COUNTRY-10 concept hierarchy generated by the standard *Chein* algorithm (left) and the *ISA recovery* algorithm (right)

Python implementation of the *Chein* algorithm

The space used by the Python solver is determined by the maximum size of the lists that implement the rectangle sets R1, R2 and R. These numbers are given in the table of Figure 30.36 at the end of each of the ten levels of the CONTRY-50 context processing. They are obtained by enabling the tracing function (see control panel of Figure 30.30). The Python solver requires 1,305 + 1,305 + 1,639 = 4,249 list elements.

	1	2	3	4	5	6	7	8	9	10
R1	50	856	1,305	955	652	407	232	119	48	10
R2	856	1,305	955	652	407	232	119	48	10	0
Disc R1	0	573	954	652	407	232	119	48	10	0
R	50	333	684	987	1,232	1,407	1,520	1,591	1,629	**1,639**

Figure 30.36 - Maximum number of elements in lists R1, R2 and R. The row **Disc R1** indicates the number of discarded elements in R1.

SQL implementation of the *Chein* algorithm

The algorithm uses three physical tables, namely R1, R2 and R, R3 being another name for R1 at the next level. The maximum size of these tables are obtained by checking the button Size of the control panel of Figure 30.31. The figures shows that the sizes are those of the Python solver, considering that table R3 plays the role of list R2 (Figure 30.37).

	1	2	3	4	5	6	7	8	9	10
R1	50	856	1,305	955	652	407	232	119	48	10
R2	1,211	254,767	455,423	220,484	90,535	31,047	8,967	1,918	162	10
R3	856	1,305	955	652	407	232	119	48	10	0
Disc R1	0	573	954	652	407	232	119	48	10	0
R	50	333	684	987	1,232	1,407	1,520	1,591	1,629	**1,639**

Figure 30.37 - Number of rows of the working tables at each level (COUNTRY-50)

The interesting result is that of table R2, which contains the rectangles created by merging the couples of R1 rectangles that match. Created by a self-join of R1 in Step S3, the space complexity of R2 is $O(|R1|^2)$. Indeed, the first levels show an important increase of the size of R2.

The execution will consume 1,305 + 455,423 + 1,305 + 1,639 = 459,672 table rows. Though the evaluation and the comparison of these figures are approximate, there is little risk to conclude that the space requirement of the SQL implementation is about *one hundred times* that of the Python implementation.

The tables of the SQL implementations (as well as that of the next algorithm) are created in an *in-memory* database so that all the three algorithms execute in RAM, which make their comparison more pertinent.

SQL implementation of the *ISA recovery* algorithm

Expressed as the query of Section 30.17, the implementation does not use any intermediate data structure. The translation of OBJ_has_ATT to CONCEPT is performed in a single step. However, the execution of a `group-by` query usually includes sorting the source table, which itself relies on its own temporary data structures, such as extracts of the table to sort and indexes. Though precisely evaluating their size would need an in-depth knowledge of the internals of the RDBMS, we can try to estimate conservative figures for the COUNTRY-50 context.

The internal `group-by` subquery processes the table OBJ_has_ATT that contains 510 rows. Sorting them requires the storage of a bit more than 510 additional temporary rows. The subquery cannot produces more than 82 rows (32 distinct attributes + 50 objects if none of them include any proper attributes). The external `group-by` query will sort and aggregate these 82 rows.

These sizes can be considered negligible.

Python implementation of the *ISA recovery* algorithm

A similar analysis will also conclude that the memory space used by the Python code is negligible.

30.11.6 Runtimes

The execution time is a critical evaluation criterion. From it, we can derive guidelines for the applicability of each technique depending on the size of datasets. We have measured the runtime of each implementation when processing the COUTRY-10 to COUTRY-60 formal contexts. These implementations do not yet include the optimization techniques we will discuss later.

The table 30.38 shows particularly dramatic results:

– When running the *Chein* algorithm, the Python version turns out to be much faster than the SQL variant. The ratio between these times seems to rapidly increase with the size of G, the set of source objects. All other parameters being equal, it appears that their complexity of the algorithm is $O(|G|^3)$, which leaves little hope for improvement[32].

– The figures relative to the *ISA recovery* algorithm (both Python and SQL implementations) just mean that it computes the concepts in less than 1 millisecond. It must be kept in mind, however, that this algorithm generates a different set of concepts than the *Chein* algorithm.

– The building of the concept hierarchy is much faster when executed by the SQL query than by the Python procedure. Python could do better: optimization desperately needed!

	Chein algorithm		ISA recovery		Hierarchy	
	Python	SQL	Python	SQL	Python	SQL
COUNTRY-10	0.02	0.07	0.00	0.00	0.13	0.03
COUNTRY-20	0.40	1.95	0.00	0.00	34.70	0.81
COUNTRY-30	3.22	15.03	0.00	0.00	702.34	6.74
COUNTRY-40	14.65	83.22	0.00	0.00	5,771.82	23.57
COUNTRY-50	45.00	359.06	0.00	0.00		54.75
COUNTRY-60	128.53	1,322.97	0.00	0.00		115.96

Figure 30.38 - Execution times of the implementations of the FCA algorithms and of the derivation of the concept hierarchy

32. See [Sarmah et al., 2013]. Also observed experimentally by a regression on the series of time values. It should be noted that the Chein algorithm, though some other algorithms have better execution times, is far from the slowest [Kuznetsov, 2001].

It may be interesting to examine in more detail the SQL implementation of the *Chein* algorithm to identify the most expensive steps. The table in Figure 30.39 details the execution times for each step of each level when solving the COUTRY-50 context[33]. It clearly shows that the step S4, which merges the rectangles of R2, is the main culprit. This is in line with the sizes of the working tables.

	1	2	3	4	5	6	7	8	9	Σ	%
S3	0.04	10.51	22.83	11.50	5.40	1.99	0.62	0.16	0.03	53.08	**14.78**
S4	0.02	42.36	194.65	57.25	9.01	1.30	0.15	0.02	0.00	304.76	**84.88**
S5	0.00	0.28	0.53	0.23	0.09	0.03	0.02	0.00	0.00	1.18	**0.33**
S6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**0.00**
S7	0.00	0.00	0.01	0.02	0.01	0.00	0.00	0.00	0.00	0.04	**0.01**
Σ	**0.06**	**53.15**	**218.02**	**69.00**	**14.51**	**3.32**	**0.79**	**0.18**	**0.03**	**359.06**	**100**

Figure 30.39 - SQL implementation of the Chein algorithm: execution time of each step of each level in seconds (solving COUNTRY-50),

30.11.7 Other techniques

Besides the Chein algorithm, the literature describes and analyzes more than a dozen FCA techniques that exhibit different performance characteristics. Although describing and comparing them would go far beyond the scope of this case study, it may be interesting to try to classify them according to different criteria. Here are three of them, which make sense after having studied the internals of the Chein algorithm.

– *Batch* or *incremental*. Batch techniques process the formal context as a whole while the incremental techniques consider each object, one at a time, integrating it to the solution built so far. The latter facilitates the evolution of the set of objects. The Chein algorithm belongs to the *batch* category.

– *Bottom-up* or *top-down*. The bottom-up algorithms start from the set of objects of the context as a first draft of the solution, then layer by layer, reduce the set of concepts to the maximal rectangles. The final solution is obtained when the last layer cannot generate new concepts. A top-down technique is an iterative procedure the starting point of which is a root concept comprising the set of all the objects of the context and all their common attributes (this latter set being often empty). Then new, more specific concepts are derived from the current state of the solution until no new concepts can be generated. The Chein algorithm belongs to the *bottom-up* category.

– *Concept set* or *concept hierarchy*. The final solution of some algorithms is a complete concept hierarchy, which comprises the set of concepts and their hierar-

33. The execution times of S1, S2 and Level 10 being less than 1 ms. are not shown.

chical organization. Other algorithms generates only the set of concepts, which is the case of the Chein algorithm. For them, the hierarchy is built by an independent procedure.

30.12 Improving the performance of the algorithms

The development of high performance implementations is not the main objective of this case study. Rather, it focuses, as do all the other studies in this series, on the modeling of the problem and of its solution(s) and on the simplicity of their implementation. Nevertheless, in some problems, the efficiency of the solution can be of crucial importance and is therefore an important part of the solving process. This is clearly the case for the FCA techniques we have designed in this chapter, some of which have impressive and, to put it bluntly, discouraging execution times.

30.12.1 *Chein* algorithm (Python implementation)

In the first implementation, the sets of rectangles R1 and R2 are represented by *lists of tuples* (g,m,d) where g is the object string-list, m the attribute string-list and d the discard indicator (Section 30.7). We ignore the representation of R (a list of couples (g,m)), common to all the techniques we will examine. We symbolize this implementation of R1 and R2 as follows:

- `R1[(g,m,d)]`
- `R2[(g,m,d)]`

The solving of contexts COUNTRY-10 to COUNTRY-60 shows much better execution times than the SQL implementation. Let us try to see whether we can gain even better results by experimenting with other implementations of sets R1 and R2. We will look at four alternative techniques.

– **2 dictionaries**. R1 and R2 are each expressed as a dictionary, symbolically denoted by `R1{g:(m,d)}` and `R2{g:(m,d)}`. A dictionary provides a quick access to an item based on the value of the key (`g`). In addition, it can be parsed as a list in the two embedded loops through the `list` function: `list(R1)`.

– **3 dictionaries + 2 lists**. In the preceding technique, the parsing of dictionary `R1{g:(m,d)}` is likely to be costly, due to its conversion into a list. We propose to complement it with a simple list `R1G[g]` of the `g` values to drive both loops. The extraction of the `m` value of `g` is then done through the dictionary, a very fast operation. The manipulation of `R2` is a bit more complex. First, we need to maintain a dictionary `R2{g:(m,d)}` and a list `R2G[g]` since, at the end of each level, `R2` becomes the new `R1`. Then, when searching `R2` for pairs of rectangles sharing the same `m` part, we need a quick access to `R2` through this `m` part, hence the additional dictionary `R2m{m:g}`. To sum up, the implementation comprises three dictionaries and two lists:

- `R1{g:(m,d)}, R1G[g]`
- `R2{g:(m,d)}, R2m{m:g}, R2G[g]`

– **5 lists**. The *2-list* implementation developed in Section 30.7 and recalled here before is simple and elegant, but does not support well the key-based access *g where m* and *m where g* since these queries must be performed by list parsing coded in Python. The idea is to let the Python engine execute them through the `index` method, likely to be faster than the equivalent Python loop. For instance, considering the list `L` and `v`, one on its element, the expression `L.index(v)` returns the position of the first instance of `v` in `L`. So, we decompose the list `R1` into three synchronized elementary lists and `R2` into two synchronized lists, totalling five lists as shown below:

- `R1G[g], R1M[m], R1D[d]`
- `R2G[g], R2M[m]`

– **2 lists of sets**. In the preceding techniques, we translated the sets of object names and of attribute names into *string-lists* (as discussed in Section 30.6.1). Now, we express these sets as Python sets of names. This could improve the union and intersection set operations. The sets of rectangles `R1` and `R2` are implemented as lists of tuples, each of them comprising a (Python) set of object names, a (Python) set of attribute names and the `d` indicator:

- `R1[(set{g},set{m},d)]`
- `R2[(set{g},set{m},d)]`

The experimental figures obtained when solving the contexts COUNTRY-10 to COUNTRY-100 are quite interesting (Figure 30.40).

The **2 dictionaries** technique (noted **2 dics**) is the most expensive and should be discarded.

The **5 lists** technique clearly appears to be the best, showing an improvement by a factor of 2.5 to 3.35 (measured for the COUNTRY-100 context, **2 dics** technique excluded).

The other three techniques, namely **2 lists**, **3 dics + 2 lists** and **sets** show intermediate figures, the initial **2 lists** technique being the more expensive, though not by far.

	2 lists	2 dics	3 dics + 2 lists	5 lists	sets
COUNTRY-10	0.02	0.02	0.01	0.01	0.00
COUNTRY-20	0.33	0.53	0.29	0.18	0.16
COUNTRY-30	2.54	6.48	2.59	1.50	1.34
COUNTRY-40	9.27	33.72	7.64	4.42	5.38
COUNTRY-50	24.12	104.96	20.56	11.10	12.35
COUNTRY-60	54.24	276.40	45.26	22.90	35.17

COUNTRY-70	121.28	751.88	97.70	46.19	80.17
COUNTRY-80	206.80	1,366.38	158.45	71.79	153.71
COUNTRY-90	481.51	3,094.88	297.64	125.13	294.77
COUNTRY-100	592.18		442.15	176.83	451.48

Figure 30.40 - Execution times (in seconds) of the five Python techniques

The graph of Figure 30.41 shows the relative position of the execution times of the five techniques.

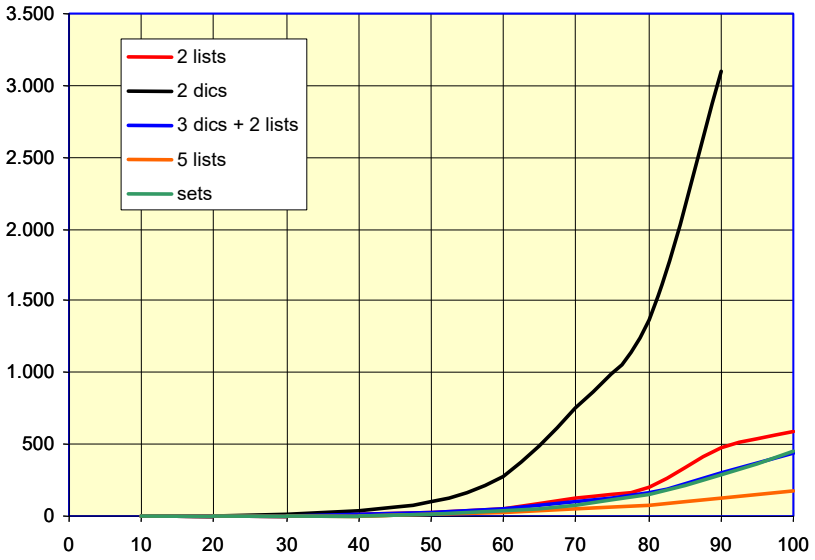

Figure 30.41 - Comparison of the execution times of the five Python techniques

30.12.2 *Chein* algorithm (SQL implementation)

The main optimization technique of relational databases is the creation of indexes. However, just as "*man proposes, God disposes*", the *database designer proposes, the RDBMS disposes*. In other words, the designer creates indexes that should logically improve the execution time of a query but in some circumstances, the SQL engine ignores the suggestion and chooses another way (hopefully better) to execute the query. This fact must be kept in mind in the following.

To try to reduce the execution time of the SQL implementation of the Chein algorithm, we analyze the structure of the queries developed in Section 30.6.2 to identify potential support indexes.[34]

34. Steps S1, S2 and S7, which each cost less than 1 ms., are ignored.

- *Step S3, Query 30.4*: the query is a self-join of table R1 based on the ordering criterion "rec1.G < rec2.G".

  Suggestion: index on R1(G)

- *Step S4, Query 30.5*: the query creates groups on column M of R2. Table R3 is filled sequentially, which does not require any index.

  Suggestion: index on R2(M)

- *Step S5, Query 30.6*: the query selects the rows of R1 that meet an equality condition on their (G,M) values. The couples of the right member of the condition are produced by an unconditional full scan of R2, which requires no index. Another way to spare access to R1 would be to store not only the G parts of the source rectangles but also their M parts, so that the equality checking can be performed *in situ* without additional access to R1. Better, since R3 is much smaller than R2, (Figure 30.37), the step S5 could be performed from the content of R3 instead of that from R2. However, the test described in Figure 30.39 shows that the contribution of step S5 to the total cost is very small, suggesting that any further optimization is worthless.

  Suggestion: index on R1(G,M)

- *Step S6, Query 30.7*: the query selects rows from R1 based on a binary column (Disc). The experiment reported in Figure 30.37 shows that the rows non discarded in R1 account for about 40% of the table. In this condition, an index on Disc would be useless and ignored by the SQL engine. Table R is sequentially filled, which does not require any index.

  Suggestion: none

To sum up, we retain these three hypotheses:

- on table R1,
  - an index on column G
  - a combined index on columns G and M
- on table R2,
  - an index on column M

To evaluate these suggestions, we apply them to the solving of contexts COUNTRY-20 to COUNTRY-60. We observe that index R2(M) brings no improvement and is discarded. On the contrary, indexes R1(G) and R1(G,M) have a positive effect that is represented in the graph of Figure 30.42. The x-axis specifies the number of source objects (20 for COUNTRY-20, 30 for COUNTRY-30, etc.) and the y-axis represents the ratio between the execution time *with index* and the time *without index*. For example, the black curve at x = 30 (COUNTRY-30) indicates that the execution time with index R1(G) is 80% (more precisely 79.6) of the time without index while the time with index R1(G,M) (the magenta curve) is slightly better (78.3%).

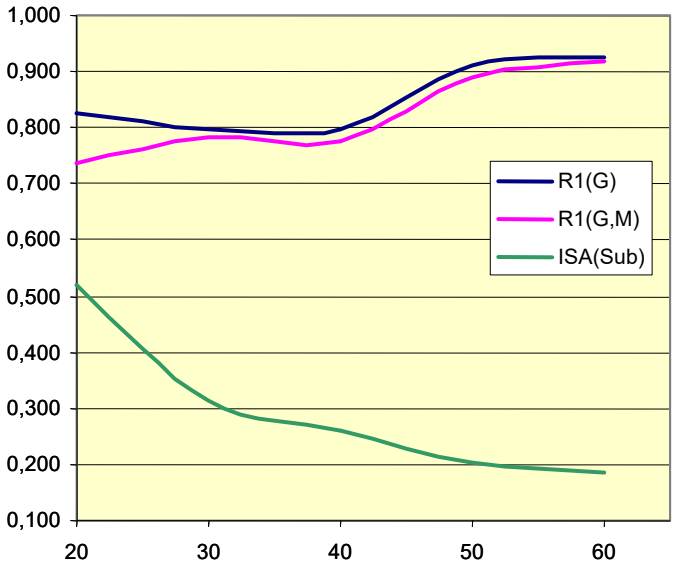

Figure 30.42 - Execution time improvement due to appropriate indexes

What does these curves tell us?

- whatever the size of the context, the improvement is not particularly impressive: from 74% to a meager 93%
- the efficiency of both indexes decreases with the size of the context
- the advantage of index R1(G,M) on R1(G) also decreases with the size of the context

To sum up, these indexes slightly improve the execution time of the SQL implementation of the Chein algorithm. However, their real usefulness is questionable, to say the least.

Note

The execution time of this SQL implementation and the influence of indexes are highly dependent on the structure (`|G|`, `|M|`, `|I|`) and content (`G`, `M`, `I`) of the source context. This can be shown by solving various configurations created by the *Context generator* (Context-generator.sql)

30.12.3 Implementation of the *ISA recovery* algorithm

The very low times measured for both the SQL and Python implementations do not suggest any specific optimization, at least for the type and size of contexts considered in this study. It would be interesting to check these times when processing larger contexts (several thousands of objects), for instance those generated by the context generator (Context-Generator.sql). This is left as an exercise.

30.12.4 Building of the concept hierarchy (SQL implementation)

We limit the discussion to the role of indexes on table ISA. The process executes two queries that create, then update, the table ISA.

The first one creates and stores in ISA the transitive closure of the inclusion relation on column G of table R (Script 30.12). It is a self-join, the join condition of which is based on a UDF (`itemInclude(sup.G,sub.G,',')`). As they are implemented in SQLfast, the UDF are *blind* to any index[35].

The second query computes the transitive reduction by deleting the transitive couples from ISA (Script 30.13). The conditions of the double self-join of the subquery includes explicit references to columns Sub and Super of table ISA, which suggests that an index on one (or on both) of these column may improve the execution time of the query. This hypothesis proves right, as the green curve of Figure 30.42 shows it:

- an index on ISA(Sub) contributes significantly to the reduction of the execution time; same for an index on ISA(Super) and ISA(Sub,Super)
- the index on ISA(Sub) proves better than those on ISA(Super) and on ISA(Sub,Super)
- the improvement is all the more important as the size of the ISA increases (times fall from 52% on COUNTRY-20 to 18% on COUNTRY-60)

Conclusion: create an index on ISA(Sub) before running the second query[36].

30.12.5 Building of the concept hierarchy (Python implementation)

The table in Figure 30.38 shows such a difference between the SQL and Python implementations that the latter seems irretrievably disqualified for a real usage.

First, it appears that the first part of the procedure (computing rawInclusion, the transitive closure of the hierarchy) is by far faster than the second (computing the transitive reduction, i.e., the final hierarchy). Thus, if an optimization is possible, it should be sought in this second part.

Let us recall its principle: we examine each couple of concepts (con1,con2) of the closure, for which we try to find an intermediate concept con3, different from con1 and con2, such that both (con1,con3) and (con3,con2) belong to the closure. If we find such a concept, then the initial couple is transitive and must be rejected, otherwise, it belongs to the reduced hierarchy.

The high cost of the algorithm seems to come from the way con3 is selected: we examine *all the concepts* (but con1 and con2, which can be neglected). Let us reduce the source of con3 to the set of superconcepts of con1. Some experiments in solving

35. In the SQL slang, such a query is called *non-sargable*, that is, non-optimizable [https://stackoverflow.com/questions/799584/what-makes-a-sql-statement-sargable]
36. Why not before the first query?

the COUNTRY family contexts show that, on the average, each concept has from 2.25 to 4 superconcepts, which is a substantial reduction in size! The code of Script 30.23 translates this idea.

```
hierarchy = []
for (con1,con2) in rawInclusion:
    status = 'basic'
    for con3 in [con for (con0,con) in rawInclusion
                 if con0 == con1 and con <> con2]:
        if (con3,con2) in rawInclusion:
            status = 'transitive'
            break
    if status == 'basic':
        hierarchy.append((con1,con2))
```

Script 30.23 - Improved procedure of transitive reduction of concept hierarchy: *reducing the search space*

The decrease of the execution times is also dramatic, as shown in the table of Figure 30.43. The figures of column Python 1 are those of the initial algorithm (copied from Figure 30.38). Those of column Python 2 are obtained by the new algorithm. The last two columns show that the SQL script still remains the best technique, specially when supported by index ISA(Sub) (column SQL 2).

	Python 1	Python 2	SQL 1	SQL 2
COUNTRY-10	0.13	0.02	0.03	0.04
COUNTRY-20	34.70	2.07	0.81	0.44
COUNTRY-30	702.34	19.73	6.74	2.16
COUNTRY-40	5,771.82	80.62	23.57	5.98
COUNTRY-50		208.99	54.75	11.60
COUNTRY-60		452.93	115.96	20.80

Figure 30.43 - Dramatic effect of simple optimizations of the concept hierarchy building (times in seconds)

30.13 On the representation of concept hierarchies

Preliminary remark

The term *Galois lattice* often appears as a synonym for *concept hierarchy*. In fact, Galois lattices are a mathematical domain at the origin of FCA. The lattice structures a set of elements among which a partial order holds. A concept hierarchy is a Galois lattice if it satisfies two properties: any pair of concepts have a unique closest (lowest) common superconcept and a unique closest (highest) common

subconcept. In particular, this implies that the hierarchy has a unique root concept and a unique leaf concept. If the hierarchy has several roots, an artificial common superconcept is added, the G part of which comprises all the objects names and the M part is empty. If the hierarchy comprises several leaf concepts, an artificial common subconcept is added, the G part of which is empty and the M part comprises all the attribute names. The various developments of this study show that not all concept hierarchies are genuine Galois lattices.

Throughout this study, we constructed various forms of equivalent hierarchy representations, in which the G part has been replaced by meaningful names, in which only proper attributes were shown, or in which all source objects have been converted into concepts. In this section, we collect all these forms (and some others) and we show how each of them derives from the others. We illustrate the discussion on the example of the ANIMAL context[37], recalled below (see Section 30.10.2):

Object	**Attributes**
CANARY	eggs,feather,fly,breath
CROCODILE	eggs,teeth,swim,breath
DUCK	eggs,feather,fly,swim,breath
FROG	eggs,swim,breath
OSTRICH	eggs,feather,breath
SALMON	eggs,swim
SHARK	eggs,teeth,swim

... and which can be represented graphically as in Figure 30.44.

CANARY	CROCODILE	DUCK	FROG	OSTRICH	SALMON	SHARK
eggs	eggs	eggs	eggs	eggs	eggs	eggs
feather	breath	feather	breath	feather	swim	teeth
fly	teeth	fly	swim	breath		swim
breath	swim	breath				
		swim				

Figure 30.44 - Graphical representation of a formal context

By application of the standard Chein algorithm, we obtain the concept hierarchy of Figure 30.45, graphically rendered in Figure 30.46.

```
(CROCODILE)x(breath,eggs,swim,teeth)
(DUCK)x(breath,eggs,feather,fly,swim)
(CANARY,DUCK)x(breath,eggs,feather,fly)
(CROCODILE,SHARK)x(eggs,swim,teeth)
```

37. Just note that these graphical representations are useful to reason on small contexts and concept hierarchies!

```
(CANARY,DUCK,OSTRICH)x(breath,eggs,feather)
(CROCODILE,DUCK,FROG)x(breath,eggs,swim)
(CANARY,CROCODILE,DUCK,FROG,OSTRICH)x(breath,eggs)
(CROCODILE,DUCK,FROG,SALMON,SHARK)x(eggs,swim)
(CANARY,CROCODILE,DUCK,FROG,OSTRICH,SALMON,SHARK)x(eggs)
```

Figure 30.45 - "GxM" representation of an FCA solution

If we feel uncomfortable with a schema in which some source object are missing, we can add them as shown in Figure 30.47 (in blue).

Since the subconcept/superconcept relation was derived from the composition of their G part (or, equivalently of their M part), the hierarchy is a redundant structure.

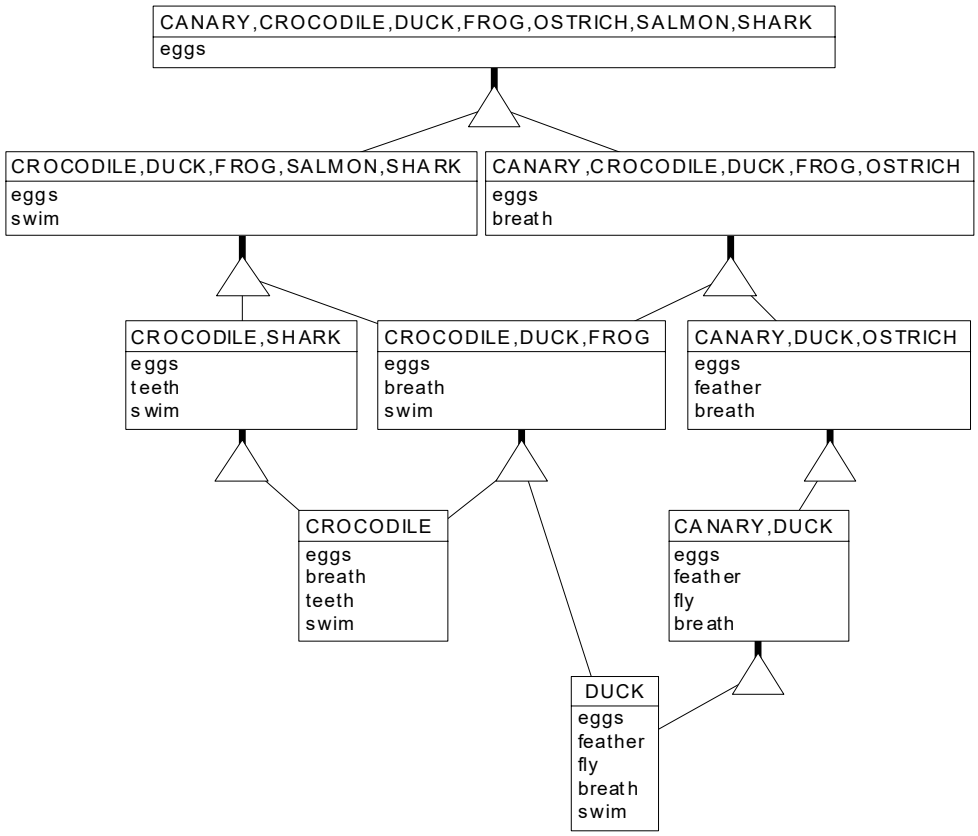

Figure 30.46 - Graphical representation of a concept hierarchy (empty source object omitted, all the attributes shown)

The reduction of the M part of the concepts to their proper attributes is shown in Figure 30.48. Again, the hierarchical structure is derivable and therefore redundant.

In some applications, such as in knowledge engineering, the nature of each concept in the real-world is important, which requires that it be given a meaningful

name. This is what we have tried to do in the schema of Figure 30.49. The hierarchy then takes the form of a taxonomy (or ontology) of a part of the real world. Since the syntactic inclusion relation between the G part of the concepts is gone, the subconcept/superconcept relation must be explicitly stated.

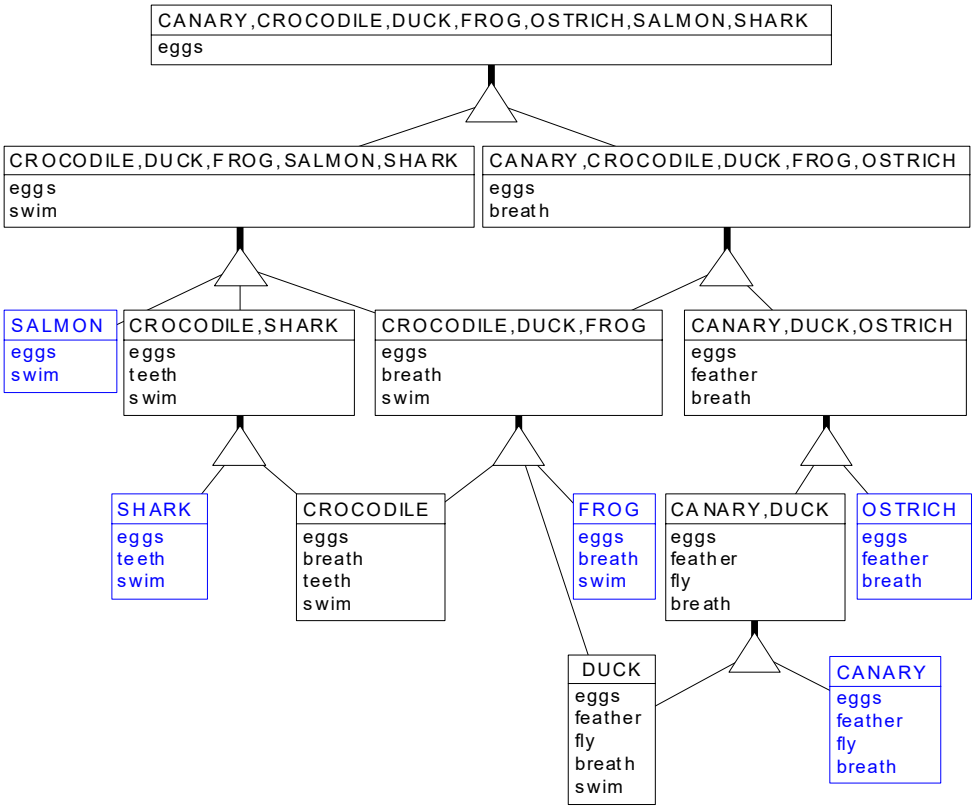

Figure 30.47 - Graphical representation of a concept hierarchy (empty source objects shown as concept)

The recovery of a complete hierarchy such as that of Figure 30.47 from the reduced version of Figure 30.48 is performed by the inheritance mechanism. Since the latter proceeds from the top of the hierarchy downward, we can call it *descending inheritance*.

Now, let us look closely at the hierarchy of Figure 30.48. We observe that the name of a subconcept (its G part) is included in the name of its superconcepts. If we remove the subconcept name from the name of its superconcept, we do not loose any information. In a sense, the superconcept *inherits* the names of its subconcepts, a mechanism that can be called *ascending inheritance*.

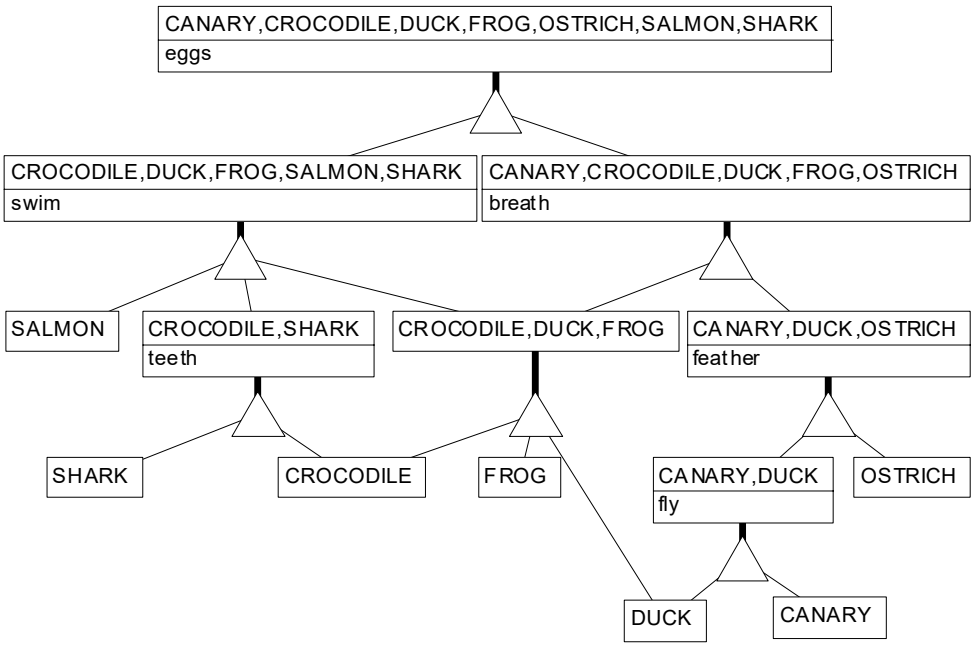

Figure 30.48 - Graphical representation of a concept hierarchy (empty source object and only proper attributes shown)

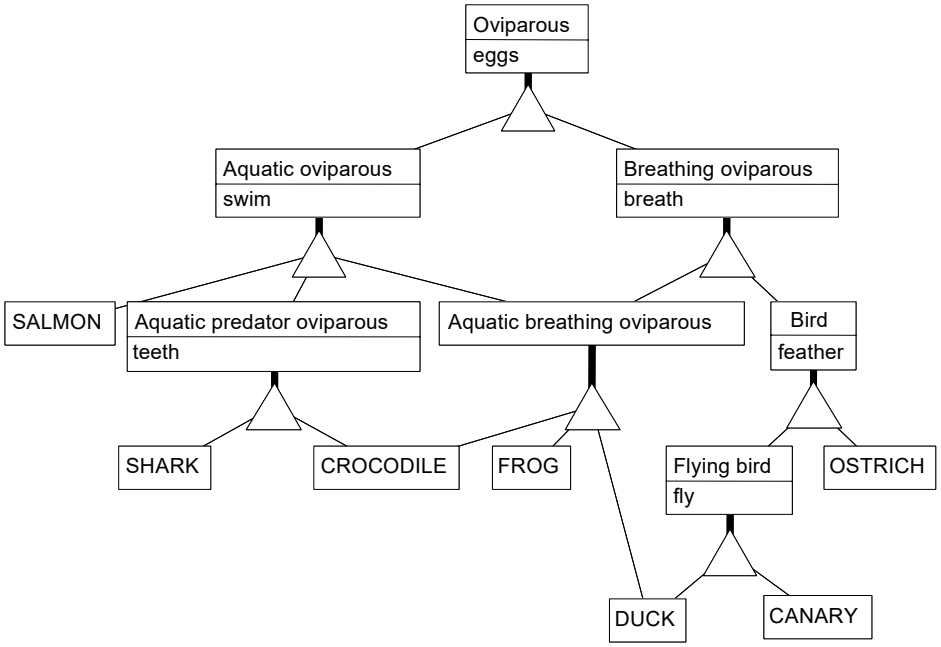

Figure 30.49 - Graphical representation of a concept hierarchy (empty source object and only proper attributes shown, G part renamed)

By applying both the descending inheritance (of the M part) and the ascending inheritance (of the G part), we get a particularly concise form of the concept hierarchy, shown in Figure 30.50. Please note that in this form, the concepts no longer are identified by their G part alone.

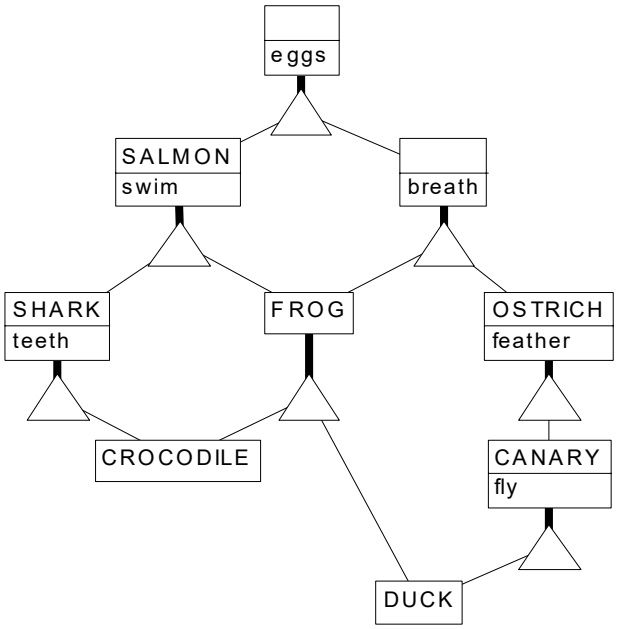

Figure 30.50 - Minimal structure of a concept hierarchy

30.14 Querying the concept hierarchy

The hierarchical structure is an appropriate support to query the concepts. Let us base our reasoning on the schema of Figure 30.35 in which *only the proper attributes* of the COUNTRY-10 hierarchy are shown. We remember that the objects are *countries* and the attributes are socio-economic properties of these countries. Considering the large number of new concepts generated (see the table of Figure 30.34), giving them meaningful names is unrealistic. Considering this hierarchy as a *database*, we can ask such questions as the following (X is a variable containing the G part of a concept):

- how many objects does concept X comprise:

```
select itemLen(G,',') from CONCEPTS where G = :X;
```

- how many subconcepts does concept X have:

```
select count(*) from ISA where Super = :X;
```

- The G part of concept C is a superset of the G part of each of its subconcepts. The larger the subconcept, the more influence it has on the composition of C. What are the most influential subconcepts of each superconcept:

```
select Super,Sub,Size
from  (select Sub,Super,
              itemLen(Sub,',') as Size,
              max(itemLen(Sub,','))
                  over (partition by Super) as MaxSize
       from   ISA
       )
where Size = MaxSize;
```

Inference rules

The analysis of the hierarchy allows us to discover hidden rules that hold in the source data. We briefly describe two of them, drawn from reference [D'Aquin, 2020].

In subconcept C, let A be a proper attribute and B an attribute inherited from one of its direct superconcept D. The descending inheritance mechanism implies that in any direct or indirect subconcept of C, the M part comprises both attributes A and B. This leads to an interesting consequence:

any object with attribute A also have attribute B,

this property can be restated as *A implies B* and formalized by the dependency rule

$A \longrightarrow B$.

To appreciate the importance of this rule, let the objects be the *countries* represented in a COUNTRY context, A be *inflation_rate:high* and B be *life_expectancy:low*. If the rule $A \longrightarrow B$ holds in the hierarchy, this means that all the countries with a *high inflation rate* also have a *low life expectancy*:

$\text{inflation\_rate:high} \longrightarrow \text{life\_expectancy:low}$

The second rule looks like, in a sense, the inverse of the first one. Again, we consider the (C<D) example discussed above. Let us call M_C the M part of C and M_D the M part of D. Let us also call S_C and S_D the sizes of C and D respectively and Mc-d the attributes of C that are not in D (proper or inherited from other superconcepts). If S_C is very close to S_D, say, with a difference of 10 percent, then we can assert that M_D implies Mc-d with a probability of 0.9 ($M_D \longrightarrow_{[0.9]}$ Mc-d). Ok, this rule seems a bit obscure, so let us apply it to the example above. We assume that concept D comprises 50 countries and concept C, 45 countries. Among the 50 countries that have a *low life expectancy*, 45 (or 90%) also have a *high inflation rate*. Hence the rule, that we will call *probabilistic dependency*:

$\text{life\_expectancy:low} \longrightarrow_{[0.9]} \text{inflation\_rate:high}$

The query of Script 30.24 computes the probabilistic dependencies that hold in the ISA table. Applied to the context COUNTRY-20, with a threshold of 0.9, it generates

the dependencies shown in Figure 30.51. We observe that this result can include transitive dependencies (here, 16 $\rightarrow_{[0.9]}$ 9). Their elimination is left as an exercise.

```
select SuperM as Left,
       '-->['||round(1.*SubSize/SuperSize,2)||']' as 'implies',
       itemExcept(SubM,SuperM,',') as Right
from (select Sub,itemLen(Sub,',')     as SubSize,
             Super,itemLen(Super,',') as SuperSize,
             C1.M as SubM,C2.M as SuperM
      from   ISA,R C1,R C2
      where  1.*SubSize/SuperSize >= 0.9
      and    ISA.Sub = C1.G and ISA.Super = C2.G
      );
```

Script 30.24 - Computing the *probabilistic* dependencies (probability 0.9)

Left	implies	Right
16	-->[0.9]	20
16	-->[0.9]	9
20	-->[0.91]	9

Figure 30.51 - The *probabilistic* dependencies in COUNTRY-20

30.15 Applications

The most general application domain of FCA is *knowledge discovery and processing* [Kuznetsov, 2004,2015], a definition that encompasses a very large variety of more specific applications. The examples used in this study suggest some typical objectives of FCA techniques, in particular:

– elicitation of hidden significant patterns, such as concepts, hierarchy of concepts and dependencies, in source data collections

– normalization of database schemas by reconstruction of subtype/supertype structures, both in the elaboration of the conceptual schema of a database under construction and in the reverse engineering of legacy databases [Beeri, 1999] [Hainaut, 2002].

The basic mechanism of FCA being to *classify* uninterpreted objects, it can be applied to any kind of these objects, such as documents and messages [Cigarrán 2016].

In software engineering, a pertinent architecture can be suggested among a collection of programs based on common resources: database tables read or updated, APIs, dialog boxes, user classes, access control (user-privilege relation).

In the database realm, besides the forward and reverse engineering mentioned above, a plausible schema can be extracted from a schema-less table (collection of rows with no common structure). Such application is common in NoSQL databases in which each record defines its own structure[38].

30.16 A short bibliography

[Beeri, 1999] C. Beeri, A. Formica, M. Missiko. *Inheritance hierarchy design in object-oriented databases*, Data & Knowledge Engineering 3 0(1999) pp.191–216

[Castellanos 2017] A. Castellanos, J. Cigarrán, A. García-Serrano. Formal concept analysis for topic detection: A clustering quality experimental analysis, Information Systems, 66(2017) pp. 24–42

[Chein, 1969] M. Chein. Algorithme de recherche des sous-matrices premières d'une matrice, Bull. Math. R.S. Roumanie, 13, 1969

[Cigarrán 2016] J. Cigarrán, A. Castellanos, A. García-Serrano. A step forward for Topic Detection in Twitter: An FCA-based approach, Expert Systems With Applications 57(2016) pp. 21–36

[D'Aquin, 2020] M. D'Aquin. A Demystifying Introduction to Formal Concept Analysis (FCA) [https://towardsdatascience.com/a-demystifying-introduction-to-formal-context-analysis-fca-ab8ce029782e]

[Hainaut, 2002] J.-L. Hainaut. Introduction to Database Reverse Engineering [https://projects.info.unamur.be/~dbm/mediawiki/index.php?title=LIBD:OUVRAGES-WEB]

[Huchard, 2000] M. Huchard, H. Dicky and H. Leblanc. Galois Lattice as a Framework to Specify Building Class Hierarchies Algorithms, RAIRO-Theor. Inf. Appl., 34 6 (2000) 521-548 [https://www.rairo-ita.org/articles/ita/pdf/2000/06/ita0110.pdf]

[Kuznetsov, 2001] S. O. Kuznetsov, S. A. Obedkov. Comparing Performance of Algorithms for Generating Concept Lattices, ICCS'01 Int'l. Workshop on Concept Lattices-based KDD pp. 35–47 [https://ceur-ws.org/Vol-42/paper3_kuznetsov.pdf]

[Kuznetsov, 2004] S. O. Kuznetsov. Machine Learning and Formal Concept Analysis, CFCA 2004, LNAI 2961, pp. 287–312, 2004. Springer-Verlag

[Kuznetsov 2015] S. O. Kuznetsov1 and A. Napoli. Formal Concept Analysis: Themes and Variations for Knowledge Processing, *Tutorial on Formal Concept Analysis at IJCAI 2015*, Buenos Aires, July 27th 2015 [https://ijcai-15.org/downloads/tutorials/T23-FCA.pdf]

38. Please note that the SQLfast environment includes a subsystem allowing users to create and exploit schema-less tables. See *SQLfast-Manual, Chapter 25 - Dynamic columns and schema-less tables*.

[Sarmah, 2013] A.K. Sarmah, S.M. Hazarika, S.K. Sinha. Formal concept analysis: current trends and directions, June 2013, Artificial Intelligence Review 44(1):1-40, June 2013 [https://www.researchgate.net/publication/255999110_Formal_concept_analysis_current_trends_and_directions]

[Wikipedia, 2023] Formal Concept Analysis [https://en.wikipedia.org/wiki/Formal_concept_analysis]