Relational Decomposition through Partial Functional Dependencies

F. Berzal[1]
berzal@decsai.ugr.es

J.C. Cubero[1] *
J.C.Cubero@decsai.ugr.es

F. Cuenca[2]
Fernando.Cuenca@xfera.com

J.M. Medina[1]
medina@decsai.ugr.es

[1] Department of Computer Science and Artificial Intelligence. University of Granada. Spain.
[2] Xfera. Spain.

Abstract

This paper introduces a new approach to database design theory. It presents the concept of partial functional dependency in the relational database framework. This new kind of dependency is an extension to the classical definition of functional dependency. The extension is accomplished following two basic ideas. First, the new dependency works over relations (not over schemes). And, second, a relation can almost verify a classical dependency in the sense that only some tuples, called exceptions, break the dependency. Throughout this work, we use that dependency to decompose relations through a more flexible and general decomposition process, and by iterating this process, we find a mechanism to extract knowledge from the original relation. Finally, we shall introduce algorithms to implement usual operations and relational system maintenance.

Keywords: Partial Functional Dependencies, Exceptions Treatment, Flexible Decomposition, Functional Dependencies.

1 Introduction

1.1 Presentation

Humans' natural way of thinking is to consider the usual form of a situation, and the problems derived from this assumption are solved ad hoc. People have a great capacity to abstract, and our suppositions are very useful to face up with daily life, even when they are false. For example, if somebody says: "Birds fly" nobody will raise doubts about it, though this assertion is false in the real world because penguins are birds but they do not fly. We deal with such problem saying that penguins are *the exceptions which prove the rule.*

*Contact Author. Work partially supported by the Spanish R&D project TIC99-0558

Such intuitive *default* reasoning is impossible to be represented in classical databases. So, this kind of knowledge cannot be incorporated in a classical relational database system directly, and the usual solution is to simplify the reality in an attempt to store it in the database. Following the preceding example, penguins will not be stored in our system. Or, on the contrary, if penguins must be represented in the database, then we must explicitly store that all birds fly, except the penguins. The main objective of this paper is to include this knowledge, based on general rules and particular exceptions to the rules, in classical databases.

The paper is structured in four sections. In section 1 we introduce the objective of our work and review previous approaches, including a general idea of the classical relational database model. Section 2 presents and discusses the formal definition of partial functional dependency. We then provide a decomposition procedure from the partial functional dependency perspective. The maintenance of such decomposition is also theoretically stated in section 3. Finally, in section 4, we discuss the design and maintenance process for a database decomposed through partial functional dependencies, from a practical point of view, and we provide precise algorithms that are necessary to implement the decomposition in a real database management system.

1.2 Background

1.2.1 Relational database model

We assume that the reader has a background in classical relational database theory, including normalization theory, so we are just going to describe the notation used through the paper (a good treatment is given in [14]).

We shall use the capital letters at the beginning of the alphabet to denote single attributes $(A, B \ldots)$, and $Dom(A)$ will be the domain of the attribute A. For sets of attributes we shall use the letters at the end of the alphabet $(X, Y \ldots)$, and $Dom(X)$ will be the cartesian product of all the attribute domains $A \in X$. $R, S \ldots$ will denote relational schemes (sets of attributes). Relations, i.e, instances of relational schemes, will be denoted by small letters such as $r, s \ldots$ and tuples by $t, u \ldots$ To emphasize that the attribute A belongs to the relation r, we shall use the notation $r.A$. The value of an attribute A in a tuple t, will be represented by $t[A]$. The symbol $\times$ stands for the usual cartesian product, and $\prod_Z(r)$ denotes the projection operator over Z applied to the relation r.

It is obvious that several relations will coexist in a database. But such relations are not completely independent, so we need to merge the information stored in several relations cohesively. In classical relational database theory, the join operator is applied to two relations, r and s, with schemes R and S, in the following way:

$$r \bowtie_\Theta s = \prod_{R \cup S} (\sigma_\Theta (r \times s))$$

Where σ_Θ is the selection operator which selects those tuples in $r \times s$ satisfying the condition given by Θ. The result of this operator is a new relation with scheme $R \cup S$. On the other hand, if r and s have X as common attributes, then the natural join of r and s is given by:

$$r \bowtie s = \prod_{R \cup (S - s.X)} (\sigma_{r.X = s.X}(r \times s)) = \prod_{(R - r.X) \cup S} (\sigma_{r.X = s.X}(r \times s))$$

1.2.2 Functional dependency and decomposition

In the previous section we have revised the classical operators to manage relations. It is important to remember that the schemes of such relations have been previously established by the database guru: this phase is the database design. The main idea behind it, is to detect semantic restrictions among the attributes and to impose its fulfillment.

The study of restrictions in databases was initiated by Codd [7], when he introduced the concept of functional dependency. In the following fifteen years, different types of dependencies were studied, such as multivalued dependencies [9], join dependencies [13], inclusion dependencies [4] and partition dependencies [8]. These works have notably contributed to database theory, but none of them has tackled the problem of exceptions. In section 1.2.3 we will revise the works that have been devoted to studying almost verified restrictions, but first, we briefly remember the concept of functional dependency (see [11] for instance).

Definition 1 *A relational scheme R satisfies a **functional dependency** (f.d from now on) denoted by $X \rightarrow Y$ with $X, Y \subseteq R$, if and only if every instance r of R satisfies the following condition:*

$$\forall t_1, t_2 \in r \quad \textit{if} \quad t_1[X] = t_2[X] \quad \textit{then} \quad t_1[Y] = t_2[Y] \quad \textit{must hold} \tag{1}$$

X contains the antecedent attributes and Y the consequent ones. It should be emphasized that functional dependencies are statements about **all** the possible relations that could be instances of a relational scheme R, and about **all** the tuples belonging to each possible instance r of R. Therefore, we cannot look at a particular relation r with scheme R and deduce what functional dependencies are verified in R, i.e., the concept of functional dependency is a semantic issue, which must be given by an expert (the database designer, usually).

It is well known that, when a relational scheme satisfies a non trivial functional dependency, there are several problems concerning updates, insertion, deletion... (see Ullman [14]). Therefore, we cannot allow a relation r to satisfy an arbitrary f.d $X \to Y$. If such a relation exists in our database (with relation scheme R), it should be decomposed into two new relations with relational schemes given by XY and $R - Y$. Such relations will be called subrelations from now on. The projection operator is used in order to merge the XY values of those tuples with equal antecedent values (and therefore, by applying definition 1, equal consequent values), then obtaining a relation called $r_2 = \prod_{XY}(r)$ with fewer tuples than r. It can be proven that this relation satisfies the f.d, and, for new entries in the database, we can test the dependency just by looking at the tuples in $\prod_{XY}(r)$. Furthermore, if we decompose the relation r into:

$$r_1 = \prod_{R-Y}(r) \quad , \quad r_2 = \prod_{XY}(r)$$

It can be proven (see [14]) that the natural join of both projections will exactly recover the original relation r itself, i.e, it is a loss less decomposition:

$$r = r_1 \bowtie r_2$$

The statement above is the most important decomposition property, because it assures that the initial information can be recovered after the decomposition. That allows the designer to decompose a relation verifying a f.d with confidence, because there will be no loss of information. This is the idea behind the process of normalization in relational databases (see [5, 6, 14]), and the purpose of this paper is to extend it to work with exceptions.

1.2.3 Horizontal decompositions

Decomposition theory has been traditionally focused on the *vertical* relation decomposition. By vertical decompositions are meant the decompositions based on the projection operator. The key point to accomplish such decompositions is the existence of a semantic restriction, so that such restriction is maintained in the system. But our approach in this paper is to accomplish both vertical and horizontal decompositions. An horizontal decomposition is a decomposition that uses the selection operator, instead of the projection one. In general, a horizontal decomposition through a f.d is accomplished using the concept of exception. The usual way to do this is relaxing the f.d in order to obtain a subrelation verifying the dependency, and isolating the exceptions to that dependency in a different relation.

Exceptions and dependencies based on exceptions have been studied, from a logical perspective, by non-classical logics (see [12]). But, from the point of view of database design, it is difficult to find works dealing with such problem. A. Borgida dealt with the problem of managing exceptions

in object-oriented databases in [1]. He provided a new kind of integrity restriction imposed over the scheme, as well as some additional semantic integrity restrictions, but his main contribution (related to exceptions) was to allow exceptions to such restrictions.

The main contributions to this field were nevertheless developed during the eighties. P. De Bra and J. Paredaens dealt with the exceptions problem, and a comprehensive treatment of them can be found in [2, 3]. In their works, different decompositions were defined to avoid *almost* verified functional dependencies in a relational scheme. The authors defined a set of exceptions with respect to a functional dependency as follows:

Definition 2 *Let r be an instance of the relational scheme R and $X, Y \subseteq R$ two sets of attributes. The relation of antecedent exceptions with respect to $X \mapsto Y$ in r is:*

$$r_{ae} = \{t \in r \quad | \quad \exists t' \in r, \quad t[X] = t'[X] \wedge t[Y] \neq t'[Y]\}$$

The semantics of this kind of exceptions is strongly attached to the antecedent values of the tuples. If a tuple is an exception, then all the tuples in the relation with the same antecedent value will be exceptions. So, we can say that the exception itself is the antecedent value. This approach is useful when there are many antecedent values verifying the dependency, and only a few ones breaking it. For example, let us consider a staff database where many employees belong to only one department, and a little set of employees belong to two or more departments. In this case, it is interesting to use this type of exceptions to isolate the employees who belong to more than one department, from the larger part that belong to only one department.

De Bra and Paredaens also introduced a normalization theory, based on *goals* and different kinds of functional dependencies. In our paper we work with exceptions from a different point of view.

2 Partial Functional Dependencies

2.1 Instance Functional Dependency

Definition 1 states that a functional dependency is defined over a relational scheme, i.e., the dependency must be exactly satisfied by all the relations of the scheme. In practice, it means that every possible relation must satisfy the f.d, that is, if a tuple does not satisfy the f.d, then it is assumed that it is a *false tuple* and can not be inserted into the relation, because, in the contrary case, we would have a relation not satisfying the f.d.

Consider, as example, a relation storing information about persons (*Identification Card, Name, Age, Sex*). The insertion of a new person with the same identification card than another one stored

in the database (a rare but possible case), is simply not allowed. What commonsense says is that almost every person should be consider as *normal* and those with an identification card equal to some other, should be consider as an *exception*. Formers, should be managed as indicated by the classical relational database theory, while the exceptions should be stored and treated in a different way.

It is obvious that if we want to relax the definition of f.d in order to consider exceptions, we will have to work with particular relations (not schemes) and study the tuples belonging to each relation.

Definition 3 *A relation instance r of a relational scheme R verifies an* ***instance functional dependency of instance*** *(i.f.d from now on) denoted by $X \mapsto Y$ with $X, Y \subseteq R$, if and only if r satisfies the condition given on equation 1.*

If we consider definition 1, it is obvious that:

A relational scheme R verifies a f.d $X \to Y$ with $X, Y \subseteq R$, if and only if every instance r of R verifies a i.f.d $X \mapsto Y$.

Definitions 1 and 3 are closely related to each other. In fact, the unique difference is that the universal quantifier (related to the relations) has been eliminated from the definition of i.f.d. This is exactly the above-mentioned idea. From now on, we will not be interested in finding semantic restrictions imposed by the designer, which must be satisfied by all the possible relations in the real world. On the contrary, the restrictions we will focus on will be data driven. To state it more precisely, the restrictions will be stated in terms of the data contained in the relation. Such idea is the main contribution of the i.f.d.

2.2 Exceptions

Informally, an exception with respect to a functional dependency in a relational database can be defined as *a tuple which breaks the functional dependency*. There are two possible ways to define the concept of exception. The first approach was introduced by P. De Bra et al. [2, 3], whose idea has been presented in section 1.2 and will be studied more thoroughly in this section. Moreover, a comparison between the previous approach and our exception definition is also introduced in this section.

From our point of view, each individual tuple is an exception as a whole. Therefore, the minimum number of exceptions is always obtained. This is a key point, because, as we shall prove, it is not true that the De Bra exceptions set is minimum in order to obtain a relation verifying a f.d. It

is important to note that this new point of view does not pretend to solve the same problems as De Bra's approach; ours is a solution to a different problem. Thus, both works cannot be directly compared. Our definition of exception is stated as follows:

Definition 4 *Let r be an instance of the relational scheme R and $X, Y \subseteq R$ two sets of attributes. We say that $r_e \subset r$ is a* ***relation of tuple exceptions*** *(or simply a exception relation) with respect to $X \mapsto Y$ in r if and only if:*

i) $(r - r_e)$ *verifies* $X \mapsto Y$.

ii) $\forall t \in r_e$, $(r - r_e) \cup \{t\}$ *does not satisfy* $X \mapsto Y$.

iii) $\nexists\, r'_e \subset r$ *verifying i) and ii) such that* $\#(r'_e) < \#(r_e)$

Remark. $\#(r)$ stands for the cardinality of r, i.e, the number of tuples which are actually stored in relation r.
Remark. If we consider that *exception* is a synonymous of *what is not common*, then, we want that r_e stores as less tuples as possible. This is the motivation of restriction *iii)*.

Roughly speaking, a set of exceptions is the minimum set of tuples that we must remove from a relation to obtain a new relation satisfying an i.f.d. If the set of exceptions were deleted from the database, then the resulting relation would satisfy a f.d and would allow us to decompose it.

Example 1 Let us consider a relation r storing information about students in a primary school. The attributes in this relation might be: *ID* (*ID* card number), *Year* (birth *Year*) and *Course*. We are interested in studying the dependency between the students' birth *Year* and the *Course* that the student is enrolled in ($Year \mapsto Course$). Clearly, the relation r never verifies a f.d between *Year* and *Course*, due to some students (usually few) who have repeated one or more courses. If the relation r in Figure 1 is considered, one can observe that students 6 and 7 have repeated once, and 8 twice. More tuples representing more students should appear in relation r, in particular other students with different ages, but they have been omitted for the purpose of focusing the reader on the interesting side of the example.

Now we compute the exceptions to the i.f.d $Year \mapsto Course$ in r. The relations in Figure 2 are the relation of antecedent exceptions (r_{ae}) and the relation of tuple exceptions (r_e).

It is interesting to study the semantics of this two different relations of exceptions generated from the same $Year \mapsto Course$ i.f.d. In r there is just one antecedent exception: the *Year* 1990; thus all the tuples corresponding to the *Year* 1990 are the exceptions following De Bra's approach and they are isolated in r_{ae}. But in r there are three tuple exceptions, because the pair of values $(1990, 4)$ is considered the usual case and therefore the tuples with *ID* values 6, 7 and 8 go to r_e. It is obvious that the tuple exceptions approach is the most suitable for this example. □

$r =$

ID	*Year*	*Course*
1	1991	3
2	1990	4
3	1990	4
4	1990	4
5	1990	4
6	1990	3
7	1990	3
8	1990	2

Figure 1: Students relation.

$r_{ae} =$

ID	*Year*	*Course*
2	1990	4
3	1990	4
4	1990	4
5	1990	4
6	1990	3
7	1990	3
8	1990	2

$r_e =$

ID	*Year*	*Course*
6	1990	3
7	1990	3
8	1990	2

Figure 2: Relations of exceptions.

It seems to be clear that the number of tuple exceptions is always lower than the number of antecedent exceptions, except for the instance relations which verify an i.f.d. In that case both r_{ae} and r_e are empty.

Proposition 1 *Let r be an instance of the relational scheme R; $X, Y \subseteq R$ two sets of attributes, and $r_e, r_{ae} \subseteq r$ two relations of tuple and antecedent exceptions, respectively, with respect to $X \mapsto Y$ in r. Then:*

$$\textit{if } r \textit{ verifies } X \mapsto Y \textit{ then } \#(r_e) = \#(r_{ae}) = 0$$
$$\textit{if } r \textit{ does not verify } X \mapsto Y \textit{ then } \#(r_e) < \#(r_{ae})$$

Proof:
When r verifies $X \mapsto Y$ it is evident that $r_e = \emptyset$ and $r_{ae} = \emptyset$. Thus $\#(r_e) = \#(r_e) = 0$.
When r does not verify $X \mapsto Y$, a non empty relation r' can be built from r_{ae} finding the largest set of tuples with equal consequents for each value in the antecedent. Then, it is easy to check that $r_e = r_{ae}$ - r' verifies *i), ii)* and *iii)* in definition 4. As r' is not empty, then $\#(r_e) < \#(r_{ae})$ ■

Due to definition 4, the set of exceptions may not be unique, so a theoretical formulation of all the possible sets of exceptions is needed. This is accomplished through the following definition:

Definition 5 *Let r be an instance of the relational scheme $R = (A_i)_{i=1\ldots n}$, I an i.f.d $X \mapsto Y$ with $X, Y \subseteq R$, and $Dom((A_i)_{i=1\ldots n})$ the domains of $(A_i)_{i=1\ldots n}$. Then we can define the operator $\mathcal{E}_I$ (called **exceptions** of r with respect to I) as follows:*

$$\mathcal{E}_I : \mathcal{P}(\times_{i=1}^{n} Dom(A_i)) \longrightarrow \mathcal{P}(\mathcal{P}(\times_{i=1}^{n} Dom(A_i)))$$
$$\mathcal{E}_I(r) = \{r_{e1}, r_{e2} \cdots r_{ek}\}$$

Where $\{r_{e1}, r_{e2} \cdots r_{ek}\}$ is the set of all the possible relations of exceptions with respect to I in r.

The exception relation is clearly not unique, and, apparently, this is a problem. But given the next proposition, we obtain an interesting property, which is very useful for our empirical point of view. Proposition 2 states that, even though the set of exceptions may not be unique, all the sets returned by the operator of exceptions have the same number of tuples.

Proposition 2 *Let r be a relation and I an i.f.d. Then, the next property is satisfied:*

$$\forall r_i, r_j \in \mathcal{E}_I(r) \quad \#(r_i) = \#(r_j)$$

Proof: Let us assume $r_i, r_j \in \mathcal{E}_I(r)$ and $\#(r_i) \neq \#(r_j)$ then $\#(r_i) < \#(r_j)$ or $\#(r_j) < \#(r_i)$ which is in contradiction with condition *iii*) of definition 4. ■

Example 2 Let r be the instance relation of Figure 1. Then, the set of all the relations of exceptions with respect to $Year \mapsto Course$ in $r' = r \cup \{9, 1991, 2\}$ is:

$$\mathcal{E}_{Year \mapsto Course}(r') = \left\{ \begin{array}{l} \{(1,1991,3),(6,1990,3),(7,1990,3),(8,1990,2)\}, \\ \{(6,1990,3),(7,1990,3),(8,1990,2),(9,1991,2)\} \end{array} \right\}$$

□

The only purpose of the operator $\mathcal{E}_I(r)$ is to formalize the definition of partial functional dependency from a theoretical perspective. In practice, we shall be interested in any relation $r_e \in \mathcal{E}_I(r)$, because our main goal is to minimize $\#(r_e)$, which is unique thanks to proposition 2. On the other hand, although it is possible that $\#(\mathcal{E}_I(r)) \neq 1$, as we have shown in example 2, this is not the typical case. Due to the exception semantics, in a real database there will be many tuples verifying the dependency for each antecedent value, and only a few tuples will be exceptions. Therefore, in practice, if the designer chooses a *good* i.f.d., then $\#(\mathcal{E}_I(r))$ will be usually 1. Otherwise the semantics of the selected i.f.d in that relation may not match with the semantics of the exceptions introduced in this work. Therefore, our solution might not be applicable to that case. This topic is thoroughly discussed in section 4.3.2.

2.3 Partial Functional Dependencies

In the previous section, the notion of relation of exceptions has been introduced. But, if we want to study the exceptions in a database relation in order to decompose such relation, a definition of partial functional dependency must be introduced. This new dependency will give significant information to the designer in order to decide which dependencies are suitable for the decomposition process.

Definition 6 *Let r be an instance of the relational scheme R, $X, Y \subseteq R$ two sets of attributes and $r_e \in \mathcal{E}_{X \mapsto Y}(r)$. Then r satisfies an α-**partial functional dependency** $X \overset{\alpha}{\dashrightarrow} Y$ (p.f.d from now on), and the α value is:*

$$\alpha = \begin{cases} 1 & if \quad \#(r) = 0 \\ 1 - \frac{\#(r_e)}{\#(r)} & otherwise \end{cases}$$

Example 3 Following example 1, we can say that the relation r verifies a p.f.d: $Year \overset{0.625}{\dashrightarrow} Course$ because:

$$\alpha = 1 - \frac{\#(r_e)}{\#(r)} = 1 - \frac{3}{8} = 0.625$$

□

In definition 6, α indicates the degree of fulfillment of the dependency between X and Y. The lower the number of exceptions r has, the greater α is. The maximum value for α is 1. In that case, both i.f.d and p.f.d definitions become equivalent:

Proposition 3 *A relation r verifies a p.f.d $X \overset{1}{\dashrightarrow} Y$ if and only if r verifies the i.f.d $X \mapsto Y$.*

Proof: r verifies $X \mapsto Y \iff \mathcal{E}_{X \to Y}(r) = \{\emptyset\} \iff \forall r_e \in \mathcal{E}_{X \to Y}(r) \quad \#(r_e) = 0 \iff \alpha = 1$ ■

Let us remark that any relation satisfies an α p.f.d. In the worst case, α is close to 0. But it is obvious that we will be interested in p.f.d with a high value (near to 1). Partial dependencies are not restrictions in the usual sense (such as functional dependencies) but they will be useful for deriving a better database design, because when α is close to 1, we shall be able to decompose the original relation (see section 3)

On the other hand, it should be emphasized that the parameter α in definition 6 is not a measure of information loss. As we shall prove, a decomposition through a p.f.d never loses information from the original relation (as in the classical case). α is just a measure of the verification level of the i.f.d in the relation, and it reports useful information to the database designer, in order to decide whether it is interesting to decompose the relation by using such dependency.

An alternative and useful way of constructing α is given by the following result:

Proposition 4 *Let r be an instance of the relational scheme R, $X, Y \subseteq R$ two sets of attributes and $X \overset{\alpha}{\dashrightarrow} Y$ an α-partial functional dependency. Then the α value is:*

$$\alpha = \frac{\sum_{x \in Dom(X)} \max_{y \in Dom(Y)} \#\{t \in r \mid t[X] = x \wedge t[Y] = y\}}{\#(r)} \tag{2}$$

Proof:
It is easy to demonstrate, due to definition 4, that the exception relation cardinal is the number of tuples of the relation minus the number of tuples verifying the dependence:

$$\#(r_e) = \#(r) - \sum_{x \in Dom(X)} \max_{y \in Dom(Y)} \#\{t \in r \mid t[X] = x \wedge t[Y] = y\} \tag{3}$$

Then, applying definition 6 and equation 3:

$$\alpha = 1 - \frac{\#(r) - \sum_{x \in Dom(X)} \max_{y \in Dom(Y)} \#\{t \in r \mid t[X] = x \wedge t[Y] = y\}}{\#(r)}$$

From the above equation, it easily follows that:

$$\alpha = \frac{\sum_{x \in Dom(X)} \max_{y \in Dom(Y)} \#\{t \in r \mid t[X] = x \wedge t[Y] = y\}}{\#(r)}$$

■

3 Decomposition through Partial Functional Dependencies

3.1 Decomposition

Once we have seen the notion of p.f.d, we proceed to introduce a more flexible decomposition method than the classical one. The main difference is that our decomposition is not only accomplished by the projection operator but the selection operator is also used.

The set of exceptions is first isolated in a different relation (r_e). This relation is obtained by applying a selection operator, so this step corresponds to the horizontal decomposition. Second, the rest of the tuples ($r - r_e$) satisfies an usual f.d and, thus, we can apply a vertical decomposition in the usual way.

Definition 7 *Let r be a relation with scheme R, $X \overset{\alpha}{\dashrightarrow} Y$ a p.f.d, and r_1, r_2, r_e three relations with schemes $R - Y$, XY and R, respectively. r_1, r_2 and r_e are a **partial decomposition** of r with respect to the p.f.d $X \overset{\alpha}{\dashrightarrow} Y$ if and only if:*

$$r_e \in \mathcal{E}_{X \mapsto Y}(r), \quad r_1 = \prod_{R-Y} (r - r_e) \quad \textit{and} \quad r_2 = \prod_{XY} (r - r_e) \tag{4}$$

The choice of r_e is arbitrary, but this is not a problem for our purposes, because, as we have said before, in a *good* decomposition (one with a high value of α) the set of exceptions should be unique. And, even in the case that the set of exceptions were not unique, we are only interested in any $r_e \in \mathcal{E}_{X \mapsto Y}(r)$ and, in practice, the most efficiently-computed relation would be chosen (as we shall show in section 4.3).

From now on, the relation r_2 will be called the *rule relation*, and tuples $t \in r_2$ will be the *rules* of the $X \overset{\alpha}{\dashrightarrow} Y$ dependency. This is because the decomposition process can be considered as an association rule mining process. The extracted rules will be $x \to y$, being x and y the values of the attributes X and Y. The objects in relation r_1 will be the objects verifying the rules, while the tuples in r_e will be the exceptions.

Example 4 Following with example 1, we can obtain the decomposition shown in figure 3 (due to equation 4). Note that the exception relation is unique in this example.

$r_1 =$

ID	*Year*
1	1991
2	1990
3	1990
4	1990
5	1990

$r_2 =$

Year	*Course*
1991	3
1990	4

$r_e =$

ID	*Year*	*Course*
6	1990	3
7	1990	3
8	1990	2

Figure 3: Decomposition through a p.f.d.

□

3.2 *MNE* Integrity Restriction

In this subsection we present the basis of the decomposition process from a theoretical point of view. That topic will be deeply treated in Chapter 4, from an empirical point of view.

Having decomposed a relation, we cannot assume that the subrelations will remain unchanged. They will be updated after the decomposition, since it will be necessary to perform insertions, deletions and even updates.

A p.f.d is not an integrity restriction itself. However, the decomposition process through a p.f.d and the subsequent maintenance raise in a natural way the idea that the exception relation must be minimal. Hence, when update operations are performed over the subrelations, it can be obtained a decomposition which does not verify the conditions stated in equation 4. In order to guarantee

that the exception set will be minimal, we define the *MNE* restriction. This restriction guarantees that the exception relation (r_e) belongs to the set $\mathcal{E}_I(r)$.

Definition 8 *Let r_1, r_2 and r_e be three relations with schemes $R-Y$, XY and R. Then r_1, r_2 and r_e verify the* ***minimum-number-of-exceptions integrity restriction*** *(*MNE *from now on), if and only if:*

$$\#(\sigma_{r_1.X=t[x]}(r_1)) \geq \#(\sigma_{r_e.XY=t[xy]}(r_e)) \quad \forall t \in \prod_{XY}(r_e) \tag{5}$$

It is important to remark that, when a relation with data is decomposed, the *MNE* restriction is verified. In this way, it can be assured that the restriction is verified before the maintenance process begins. So, if the update operations are defined in such a way that they maintain the *MNE* restriction, then, the decomposition will always verify such restriction.

Proposition 5 *A partial decomposition, obtained by equation 4, verifies the* MNE *restriction.*

Proof: From equation 4, let us assume that $\exists t \in \prod_{XY}(r_e)$ which does not verify $\#(\sigma_{r_1.X=t[x]}(r_1)) \geq \#(\sigma_{r_e.XY=t[xy]}(r_e))$. Then, we can construct three relations r'_1, r'_2 and r'_e in the following way:

1. $r'_1 = r_1$, $r'_2 = r_2$ and $r'_e = r_e$.
2. Move all the tuples $\sigma_{r_2.X=t[x]}(r_1 \bowtie r_2)$ into r'_e.
3. Move all the tuples $\prod_{ZX}(\sigma_{r_e.XY=XY[xy]}(r_e))$ into r'_1.
4. Insert the tuple $t[xy]$ into r'_2.

Thus, all the conditions stated in definition 4 are verified except condition *iii)* because $\#(r'_e) < \#(r_e)$. Thereby, $r_e \notin \mathcal{E}_{X \mapsto Y}(r)$, which is in contradiction with the initial assumption. ■

The preceding proof, besides proving proposition 5, shows the way to enforce the *MNE* restriction in a partial decomposition. The *Local_Redistribution* algorithm introduced in section 4, performs these steps.

3.3 Partial Natural Join

A procedure to accomplish the partial decomposition has been presented in the previous subsection. From a theoretical point of view, the *MNE* restriction guarantees the semantic restriction imposed by a p.f.d during the maintenance phase. As in the classical case, it is also necessary to *combine* the information contained in the subrelations into one relation. We introduce the partial

natural join operator in order to accomplish this task. As the partial decomposition is both horizontal and vertical, it is obvious that two operations are needed. The classical natural join will be used for the vertical component, and the classical union operator for the horizontal one.

Definition 9 *Let r be a relation with scheme R,and let r_1, r_2 and r_e be the relations obtained by equation 4. The* ***partial natural join*** *of r_1, r_2 and r_e denoted by $\mathcal{J}(r_1, r_2, r_e)$ is a relation with a scheme R, and it is obtained in the following way:*

$$\mathcal{J}(r_1, r_2, r_e) = (r_1 \bowtie r_2) \cup r_e$$

The partial natural join operator can recover the same relation we had before the decomposition. Recovering the information we had at the beginning of this process is an issue as important as decomposing a relation and obtaining various relations with interesting properties. This is the most important property of the classical decomposition, and it is said that the decomposition is *loss less*. We prove now that the partial decomposition is also loss-less.

Theorem 1 *Let us assume the same conditions as in definition 9. Then $r = \mathcal{J}(r_1, r_2, r_e)$, i.e., the decomposition of r through r_1, r_2 and r_e is loss less.*

Proof: Let us assume r satisfying a p.f.d $X \overset{\alpha}{\dashrightarrow} Y$ with an arbitrary α. We can write $r = r_e \cup \{r - r_e\}$. Now, $(r - r_e)$ satisfies an i.f.d $X \mapsto Y$, as equation 1 holds. Thus, by Heath's theorem [10], the decomposition of $r - r_e$ in $\prod_{XY}(r - r_e)$ and $\prod_{R-Y}(r - r_e)$ is loss-less, that is:

$$r - r_e = \prod_{XY}(r - r_e) \bowtie \prod_{R-Y}(r - r_e) \Longrightarrow \left(\prod_{XY}(r - r_e) \bowtie \prod_{R-Y}(r - r_e)\right) \cup r_e = r$$

■

Let us emphasize that theorem 1 states that the α degree of a dependency does not imply a loss of information because we can recover exactly the same information we originally had in relation r.

3.4 Successive decompositions

When a p.f.d is used to decompose a relation, three new relations are obtained. If those relations do not satisfy any dependency, then the decomposition process has finished. Otherwise, they could be properly decomposed depending on the kind of dependency they satisfy (functional, partial, etc). Partial decomposition should not be seen as an independent process unrelated to classical normalization. The following example shows how to combine decompositions through f.d, p.f.d, and others in order to obtain a good final scheme.

Example 5 In a ship booking database, we want to manage the reservation of passengers' pets (dogs and cats, mainly). The system stores for each pet: its *ID* (the number in the *ID* card), its *Size* (three-valued: *Big*, *Medium* or *Small*), its *Cage* type (type *A* or *B* only) and *Fare*. The following rules must be considered in order to guarantee that the animals are well accommodated:

1. Each cage type has an specific size: type A is for *Small* and *Medium*-sized animals, while type B is for the *Big* ones.
2. The cage fare is: 50$ (type A) or 70$ (type B).
3. An animal might be moved from an A cage to a B cage if the pet owner demands it or even if special characteristics of the animal require it (if it is very nervous for example). In this case, the owner should pay the type B fare.
4. If an animal had to be fitted in a cage of type A but there were not any free cages of this type, then the animal would be assigned to a cage of type B still paying the fare corresponding to the type A.

$r =$

ID	*Size*	*Cage*	*Fare*
1	*Small*	*A*	50.00$
2	*Big*	*B*	70.00$
3	*Medium*	*A*	50.00$
4	*Small*	*A*	50.00$
5	*Medium*	*B*	50.00$
6	*Medium*	*B*	70.00$
7	*Big*	*B*	70.00$

Figure 4: Reservation database.

Let us assume the relation r shown in Figure 4. That relation does not verify any f.d, and therefore we cannot normalize it. But if the p.f.d $Size \overset{\frac{5}{7}}{\dashrightarrow} (Cage, Fare)$ is considered, the relation can be decomposed with a high level of dependency verification, as shown in Figure 5. Now there are no exceptions left in r_2, and thus a classical f.d $Cage \rightarrow Fare$ is satisfied. Therefore, it can be further decomposed in the classical way, constructing the relations $r_{21} = \prod_{Size,Cage}(r_2)$ and $r_{22} = \prod_{Cage,Fare}(r_2)$. Finally the original relation can be recovered by computing $\mathcal{J}(r_1, r_{21} \bowtie r_{22}, r_e)$, as indicated in Figure 6.

In the final scheme, the information is distributed as follows:

$r_1 =$

ID	*Size*
1	*Small*
2	*Big*
3	*Medium*
4	*Small*
7	*Big*

$r_2 =$

Size	*Cage*	*Fare*
Small	*A*	50.00$
Medium	*A*	50.00$
Big	*B*	70.00$

$r_e =$

ID	*Size*	*Cage*	*Fare*
5	*Medium*	*B*	50.00$
6	*Medium*	*B*	70.00$

Figure 5: First decomposition: r_1, r_2, r_e

$r = r_1 \bowtie \left((r_{21}) \right.$

Size	*Cage*
Small	*A*
Medium	*A*
Big	*B*

$\bowtie$ (r_{22})

Cage	*Fare*
A	50.00$
B	70.00$

$\left. \right) \cup r_e$

Figure 6: Final decomposition: r_1, r_{21}, r_{22}, r_e

- r_1 stores usual reservations.
- r_{21} contains which kind of cage is suitable for each size.
- r_{22} includes the cage fares.
- r_e includes the special reservations (the exceptional cases).

□

3.5 Iterated decompositions

The importance of applying successive decompositions has been introduced in the previous subsection, but this process will be even more interesting if an exception relation is decomposed with respect to the same p.f.d which generated it (using a different parameter α). This yields an iterated decomposition process where a new decomposition level is generated in each step.

Briefly speaking, we first consider a p.f.d $X \overset{\alpha}{\dashrightarrow} Y$ over a relation r, the corresponding exception relation r_e, and the relations r_1 and r_2 derived by decomposition. This would be the first level. Now,

we can apply again the same p.f.d to r_e (with a different value of α). Thus, by decomposing r_e, we obtain a second set of relations, i.e, the second level. This will be called the *several level approach*. The process continues until no exceptions are found in a level or a stop condition (imposed by the expert) is met. One example of such condition is setting a maximum level (see example 6, for instance).

We can choose an alternative representation for the levels generated in the previous process. A new attribute can be added to the initial scheme, which will be called *Level*, and will store the level the tuple belongs to. Therefore, the original relation can be decomposed using the f.d $(Level, X) \rightarrow Y$. This new representation for the subrelations only changes the way the data is stored, but it does not change the global amount of information. The main virtue of this new representation is that we obtain two subrelations where we had $2 \cdot N$ before (being N the number of levels). This is the *level attribute approach*.

The whole process will be clarified in the next example:

Example 6 Given the relation r shown in Figure 1, with the same conditions as in Example 1, let us first consider the *several level approach*. If r is decomposed through the p.f.d $Year \overset{5/8}{\dashrightarrow} Course$, an exception relation $r_e = \{(6, 1990, 3), (7, 1990, 3), (8, 1990, 2)\}$ will be obtained, as stated in Figure 3. Relation r_e refers to those students who have repeated some courses. Now r_e can be further decomposed into three new relations using the p.f.d $Year \overset{2/3}{\dashrightarrow} Course$ again. This process can be iterated as many times as we want. But if we suppose that a student is not allowed to repeat more than two courses in the same school, the exception relation for the second decomposition verifies a p.f.d $Year \overset{1}{\dashrightarrow} Course$ (and due to proposition 3, it also verifies an i.f.d $Year \mapsto Course$), so that the exception relation can be eliminated as shown in the left side of Figure 7.

A three-level decomposition is obtained when the previous process is over. Each level corresponds to a set of students sharing one characteristic: they have repeated the same number of courses. It is important to note that this information is not present at the original relation, but it has appeared during the decomposition process through the p.f.d, that is, we have discovered hidden information.

Let us now consider the *level attribute approach*. We add a new attribute (called *Level*) to the original relation r, obtaining a relation called r'. This attribute stores the tuple level, i.e, the number of courses the student has repeated. Now, r' can be decomposed through a classical f.d $(Year, Level) \rightarrow Course$, as can be seen in Figure 8. Note that the *Level* attribute is automatically computed by the system, it is not new information the user needs to introduce.

□

$r =$

ID	*Year*
1	1991
2	1990
3	1990
4	1990
5	1990

$\bowtie$

Year	*Course*
1991	3
1990	4

$\cup$

ID	*Year*
6	1990
7	1990

$\bowtie$

Year	*Course*
1990	3

$\cup$

ID	*Year*
8	1990

$\bowtie$

Year	*Course*
1990	2

Figure 7: Iterated decompositions.

$r' =$

ID	*Year*	*Level*
1	1991	0
2	1990	0
3	1990	0
4	1990	0
5	1990	0
6	1990	1
7	1990	1
8	1990	2

$\bowtie$

Year	*Level*	*Course*
1991	0	3
1990	0	4
1990	1	3
1990	2	2

Figure 8: Decomposed r' relation.

4 Partial Functional Dependencies: Implementation

In the previous sections we have introduced the notion of p.f.d from a theoretical point of view. Now, we face the problem of implementing the decomposition process in a relational database management system (RDBMS from now on). We shall define the usual operations performed on a relation and we shall treat the problem of maintaining the integrity in relations decomposed by a p.f.d.

4.1 P.f.d. Decomposition process

The usual way to accomplish the design process is the following: first, the designer proposes a scheme and establishes the dependencies which should be verified, and then, with the help of the normalization theory, he(she) appropriately decomposes such scheme. This is an *a priori* approach, in the sense that there are no tuples in the relations. Functional dependencies are restrictions imposed in the database by the designer and must agree with the reality he wants to model. Alternatively, we can follow an *a posteriori* approach and proceed to decompose a relation which already exists in our database. So, attending to the moment when the decomposition takes place, we distinguish the following situations:

1. *A priori* decomposition: This is the counterpart of the classical design process. The designer decides if it is interesting to use a p.f.d to decompose a relation before introducing data. This decision is based on the knowledge he has about the world. At this step, there is not a concrete value of α but, according to the designer criteria, it will be close to 1 after sufficient data has been introduced.

2. *A posteriori decomposition:* This is a completely different approach. Now, the scheme has been a priori fixed by the designer, and we are faced with relations containing data. Partial dependencies are now obtained by a discovery process working on the existing tuples. Definition 6 is used to compute the α level of verification of the dependency, and then the designer chooses to decompose a relation or not according to the value of α.

The *a priori* decomposition may be used when the designer is completely sure that there exists *almost* a functional dependency among several attributes, i.e, there are not too many exceptions, and so a partial funcional dependency applies. On the other hand, the *a posteriori* decomposition provides more reliability about the quality of the decomposed scheme. If we have a relation with enough data, in the sense that they are a good representation of the reality, then we can assure a good behaviour of the decomposition.

4.2 Accessing to decomposed relations

Once the designer decomposes a relation (through an *a priori* study or via an *a posteriori* approach) he must cope with three new relations which replace the old one. Then, he must settle the update operations on these relations. In the classical case, the modifications are easy to perform because each relation is independent of the other ones, i.e, updates on one relation do not affect the others. The same does not apply when working with decompositions stemming from a p.f.d. There are two approaches:

1. **Direct update:** In this case, the user has direct access to each relation and can update them. In most cases, the database administrator should establish a particular update policy

because a modification in one relation could force modifications to other relations. Let us give an example to clarify this idea.

Example 7 Let us consider example 4 again. If we update the tuple $(7, 1990, 3)$ in r_e, and set it to $(7, 1990, 4)$, then we have a student who has passed his exams and, thus, he is not an exception. So we have to delete $(7, 1990, 3)$ from r_e and insert $(7, 1990)$ into r_1; r_2 is not modified.

On the other hand, let us suppose we want to change the tuple $(1990, 4)$ in r_2 to $(1990, 5)$. What do we have to do? The answer is simple: it depends on the semantics of this modification. For instance, it could mean that these students are automatically passing to the next course. Or it could mean that there has been an important change in the education authorities policy, and now all the students begin their school days one year younger: in this case, the p.f.d changes because we now have students following the *old* syllabus and other students with the *new* one in the same school.

In any case, one thing is clear: updates to r_2 are not straightforward, and non-expert users should be forbidden to update it by the database administrator. □

2. **Update through an *universal relation*:** In this case, all the updates are defined through a fictitious relation which is called the universal relation. A thorough study of this concept can be found in the Ullman's textbook [15]. This virtual relation is a view built from the relations obtained after the decomposition, through the partial natural join operator (see Figure 9). The user interacts with this relation as he would do with the original one, and the system is in charge of the corresponding modifications.

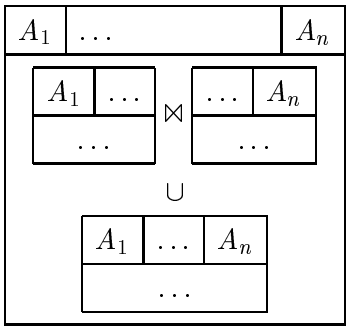

Figure 9: Universal relation.

As we have mentioned, the concept of universal relation can be implemented in a RDBMS through a view. If, in addition, the RDBMS allows views on views, then we can construct the universal relation related to an iterated decomposition.

The main objective of our work is to cope with the second approach (through the universal relation) because the first one is completely problem-oriented. So, if we are going to use the three relations (r_1, r_2, r_e) as if they were one, what is the advantage offered by the decomposition? These are some possible answers:

- *Flexible restrictions:* In a classical database, the only way to decompose is imposing a restriction as strict as a f.d. If the database administrator does not want to decompose a scheme, then he is forced to manually implement any kind of restriction he wants to impose (using stored procedures, for instance).

  If we use partial dependencies we allow the definition of flexible restrictions, i.e, restrictions which do not affect all the tuples. An apparently drawback of this approach is that any tuple which breaks the p.f.d is an exception and can be accepted and stored in r_e by the system. But, as we shall see in Section 4.5, we can use several mechanisms to control the *quality* of such exceptions. In other words, some exceptions could be rejected as valid tuples and would not be stored.

- *Default information:* When we decompose r into r_1, r_2 and r_e through a p.f.d $X \overset{\alpha}{\dashrightarrow} Y$, relation r_2 stores the rules corresponding to the i.f.d $X \mapsto Y$ existing in $r - r_e$. Such knowledge may be used to provide a friendly interface to the user by automatically filling the consequent values with the default ones (those appearing in the rule). The user could change those values when an exception comes into the system.

  Example 8 Let us consider example 5. If we want to construct a data entry interface we can use a master/detail form, with r_1 as the master relation and r_2, r_{22} as details. When the user inputs the animal size, the system would automatically fill the values for *Cage* and *Fare* according to the rules in r_2. If an exception comes into the system, the user only has to change these values by the right ones. Then, the system would transfer this tuple to the exception relation r_e. □

- *Database compression:* It is obvious that the scheme derived from a partial decomposition needs less storage space than the original one, and this is always a worthy achievement.

- *Better understanding of the database:* The new scheme is an abstraction of the stored data, and it will be easier to understand because it works with smaller relations which have a more precise semantics.

- *Direct access:* The usual way to access the decomposed subrelations is through an universal relation, but direct access to them should be allowed to the database administrator. This direct access increases efficiency and simplicity when manipulating those relations. This improvement applies to r_2 in particular, which has a primary key upon which we can create an index.

The only disadvantage of this approach is that the access to the exceptions is somewhat slower now, but, as they are sparse, this is not a problem if we compare it with the benefits obtained with r_1 and r_2.

4.3 Update Operations

We proceed now to introduce specific algorithms to perform the usual update operations when working with the universal relation, no matter the schema semantics (world modelling) is. In the case of direct access to the decomposed relations, the update operations are done as always, although they are restricted by the database administrator according to the problem semantics.

When a relation has been iteratively decomposed, the update operations are recursively performed with the same algorithms. For instance, during a tuple insertion process into the universal relation r, we could have to perform an insertion into r_2. If this relation had been decomposed by a p.f.d, then we should have applied the same insertion algorithm recursively for partial decompositions.

The algorithms we are going to introduce are described in a pseudo programming language based on Oracle PL/SQL and Pascal, but they are not restricted to working with a given RDBMS at all. The RDBMS should meet some minimum requirements however:

- *Stored procedures:* The RDBMS should allow programming functional procedures. They are usually stored in the database itself, they have direct access to the relations and they are usually optimized for usual data management operations (OLTP processing).
- *Exception manager:* An exception manager executes a special-purpose pcode fragment when an abnormal situation occurs while trying to execute a procedure. This is usually an error or an attempt to violate the database integrity.

  The reader should note that this programming technique has nothing in common with the concept of exception we are treating in this work. The context will be enough to distinguish between both concepts.

From now on, we assume that we are working with the relations r_1, r_2 and r_e, which have been derived by applying a partial decomposition to r, with scheme $R = ZXY$, through a p.f.d. $X \overset{\alpha}{\dashrightarrow} Y$. As r_2 verifies the i.f.d. $X \mapsto Y$, then we can impose the primary key restriction to the set X in r_2 .

4.3.1 Insertion

In the following sections we present the algorithms needed to maintain the partial decomposition. The first one is the insertion algorithm. We shall introduce a first version now and we shall give the definitive one later. The preliminary version of the insertion algorithm is presented in Figure 10. This algorithm is executed when a new tuple t is inserted into the universal relation r: it tries to insert t into r_1 (the XY-values) and r_2 (the ZX-values); if t is an exception, then the primary key restriction over r_2 would be violated and, thus, the tuple would be inserted into r_e.

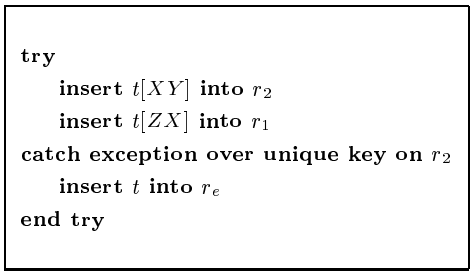

Figure 10: First version of the insertion algorithm.

The execution of the *try-catch* block is as follows: the RDBMS tries to execute those sentences between the reserved words *try* and *catch*. If an exception occurs during the execution, then there is a jump (thus, the *try* block finishes at this point), and the system executes those sentences between *catch* and *end-try*. There could be several types of exceptions (several catch blocks), but the *unique key* exception, which is thrown when the primary key restriction imposed on r_2 is violated, is the only one we need for our purposes.

Let us now see what the problem with the algorithm given in Figure 10 is. The natural join of r_1 and r_2 satisfies the i.f.d. $X \mapsto Y$, but r_e could violate the *MNE* restriction. The number of exceptions generated by this algorithm depends on the tuple insertion order, as can be seen in the following example.

Example 9 Let r be a relation with no tuples and scheme (A, B, C), and let us consider the next sequence of database insertions: (a_1, b_1, c_3), (a_2, b_2, c_2), (a_3, b_1, c_1) y (a_4, b_1, c_1). Let us now consider the p.f.d. $B \overset{1}{\dashrightarrow} C$, and the corresponding decomposition. Then, the insertion algorithm obtains the relations in Figure 11. But it is easy to see that the right decomposition in order to maintain the *MNE* restriction, would be the one shown in Figure 12.

□

This example shows that it is necessary to check the *MNE* restriction after each insertion, so that the algorithm properly modifies the relations r_1, r_2 and r_e when that restriction is not verified. We

$r_1 =$

A	B
a_1	b_1
a_2	b_2

$r_2 =$

B	C
b_1	c_3
b_2	c_2

$r_e =$

A	B	C
a_3	b_1	c_1
a_4	b_1	c_1

Figure 11: Erroneous insertion obtained with the first version of the algorithm

$r_1 =$

A	B
a_2	b_2
a_3	b_1
a_4	b_1

$r_2 =$

B	C
b_1	c_1
b_2	c_2

$r_e =$

A	B	C
a_1	b_1	c_3

Figure 12: Right insertion

present the correct version of the insertion algorithm in the next section.

4.3.2 Redistribution

The operation in charge of transforming the relations which stem from the decomposition, so that the *MNE* restriction holds, is called **redistribution**. Each time a new tuple is appended to the universal relation, it is classified as *normal* if it goes to r_1 and r_2, or as an *exception* if it is sent to r_e. When this classification is not correct because the *MNE* restriction is not verified, then it is necessary to redistribute the normal and the exceptional tuples. The algorithm given in Figure 13 (called *Local_Redistribution*) performs this task, maintaining the *MNE* restriction and, thus, solving the problem we found in the first version of the insertion algorithm.

The algorithm described in Figure 13 does not check the *MNE* restriction for all the tuples (and that is the reason why it is called *local*). It only tests if the tuples t with $t[X] = x$ are correctly distributed. This verification is enough for our purposes because the insertion of a tuple t cannot alter the classification of those tuples with a different X-value.

Now, we can update the insertion algorithm given in Figure 10 so that it maintains the *MNE* restriction, just by calling *Local_Redistribution* as shown in Figure 14.

A main drawback of this second version is that maintaining consistency (*MNE* restriction) by reconstructing the universal relation view at update time may reduce performance dramatically, specially when there are too many redistributions. So, it could be interesting to allow r_e not to verify the restriction, at least for short periods of time. We are going to consider several ways to cope with the problem of maintaining the *MNE* restriction, grouped in three categories:

```
algorithm Local_Redistribution (
    r1, r2, re: relations stemming from a p.f.d. decomposition (X -α-> Y)
    xy: a pair of XY values)
begin
    nr1 = #(σ_{r1.X=x}(r1))
    nre = #(σ_{re.XY=xy}(re))
    if (nr1 < nre) then /* Redistribution */
        insert σ_{r2.X=x}(r1 ⋈ r2) into re
        delete from r1 where r1.X = x
        delete from r2 where r2.X = x
        insert ∏_{ZX}(σ_{re.XY=xy}(re)) into r1
        insert (xy) into r2
    end if
end algorithm
```

Figure 13: *Local_Redistribution* Algorithm

```
algorithm Insertion (
    r1, r2, re: relations stemming from a p.f.d. decomposition (X -α-> Y)
    t: tuple to be inserted)
begin
    try
        insert t[XY] into r2
        insert t[ZX] into r1
    catch exception over unique key on r2
        insert t into re
        Local_Redistribution (r1, r2, re, t[XY])
    end try
end algorithm
```

Figure 14: Second version for the insertion algorithm.

- **Dynamic Mode**
  - *Maintaining the* MNE *restriction always:* As in Figure 14, *Local_Redistribution* is called after each update operation.
  - *Enforcing the* MNE *restriction periodically:* As we have mentioned, the previous approach could be inefficient, especially at the beginning of the data insertion, when there are only a few tuples and it is not clear which ones the exceptions are, so there will be redistributions frequently. One possible solution is not to maintain the *MNE* restriction all the time. When a given criterion is met (number of tuples, time elapsed from the last insertion, etc), a local redistribution will be applied to all the xy values in r_e. This task is performed by the *Global_Redistribution* algorithm described in Figure 15.

```
algorithm Global_Redistribution (
    r1, r2, re: relations stemming from a p.f.d. decomposition (X -α-> Y)
begin
    for each tuple t in ∏XY(re) do
        Local_Redistribution (r1, r2, re, t)
end algorithm
```

Figure 15: *Global_Redistribution* algorithm.

- **Static Mode**

  In this mode, we consider that the rules in r_2 are valid and cannot be automatically modified by the system, i.e, insertions of new tuples into the universal relation cannot modify the existing tuples in r_2. So, a redistribution never will be applied in this mode, and, thus, the *MNE* restriction could be violated.

  Where do the rules come from in the static mode? Obviously, the database designer is the person who has to provide them. There are two possible sources:

  – The tuples in r_2 can be established by the expert attending to the domain knowledge he has about the world which is been modelling. The database designer could change these tuples whenever he wanted, and no automatic redistribution would be performed. A normal user should not have permission to modify this relation.

  – Another scenario takes place when we have a non-decomposed relation with enough data. Then, if the database designer agrees, it can be decomposed through a p.f.d. The tuples in r_2 are the induced rules which can be discovered from the current data, which might be manually altered by an human expert.

- **Hybrid Mode**

  The previous two modes can be combined into a more flexible one, guided by the database designer. At the beginning of the tuple insertion into in the universal relation, dynamic mode should be used, so that the rules in r_2 would be tuned by the redistribution algorithm. Once there are enough tuples in the database, the system turns into static (safe) mode, so that the previously generated rules are kept with no modification. Whenever the database designer wants it, the system can turn again into dynamic mode (if he considers that the rules governing the world he is modelling have changed, for instance).

Taking into account all the possible modes, we can give the general redistribution algorithm in Figure 16, and the definitive insertion algorithm shown in Figure 17. We suppose there are

two global variables, *mode* and *MNEcriterion*, which are changed by the designer and control the redistribution mode.

```
algorithm Redistribution (
    r1, r2, re: relations stemming from a p.f.d. decomposition (X -α-> Y)
    xy: a pair of XY values)
begin
    case mode of
        Static: /* Do nothing */
        Dynamic: /* Redistribution if necessary */
            case MNEcriterion of
                Always:
                    Local_Redistribution (r1, r2, re, xy)
                Periodically:
                    if Global_Criterion() then
                        Global_Redistribution (r1, r2, re)
                    end if
            end case
        end case
end algorithm
```

Figure 16: *Redistribution algorithm*

```
algorithm Insertion (
    r1, r2, re: relations stemming from a p.f.d. decomposition (X -α-> Y)
    t: tuple to be inserted)
begin
    try
        insert t[XY] into r2
        insert t[ZX] into r1
    catch exception over unique key on r2
        insert t into re
        Redistribution (r1, r2, re, t[XY])
    end try
end algorithm
```

Figure 17: Final version of the insertion algorithm.

4.3.3 Deletion

When deleting tuples, happened during insertion, it may be necessary to perform a redistribution. The algorithm shown in Figure 18 deletes a tuple from the universal relation, and then calls to *Redistribution*. The most important remark about the algorithm in Figure 18 is that a deletion of

a non-exception tuple is accomplished by just deleting the tuple $t[ZX]$ in r_1. A rule $t[XY]$ in r_2 will only be deleted when there is no tuple t' in r_1 with $t'[X] = t[X]$.

```
algorithm Deletion (
    r1, r2, re: relations stemming from the decomposition with a p.f.d. X -α-> Y
    t: tuple to be deleted)
begin
    if t ∈ re then
        delete t from re
    else
        delete t[ZX] from r1
        if (#(σ_{t[X]=r1.x}(r1)) == 0) then
            delete t[XY] from r2
        end if
        Redistribution (r1, r2, re, t[XY])
    end if
end algorithm
```

Figure 18: Deletion algorithm

4.4 Efficiency

We have shown the benefits the partial decomposition provides in the design of a scheme satisfying a functional dependency with exceptions. However, it would be of no interest if an inefficient access to the relations were the price to pay. When the user is accessing directly to the decomposed relations, the improvement is obvious because there are fewer tuples, and, besides, we can define an index over r_2. So, we will focus on the universal relation approach. Let us consider the worst case, which happens when the dynamic mode strictly maintains the *MNE* restriction.

All the simple operations appearing in the algorithms (such as *select*, #, *delete* and *insert*) can be performed by any RDBMS through the typical SQL statements and operators *SELECT*, *COUNT*, *DELETE* and *INSERT*, which are highly optimized, usually $O(\log \#(r_e)))$.

On the other hand, the unique key restriction check is automatically performed by the RDBMS system, so that it maintains the database integrity. If we do not take into account the calls to *Local_Redistribution* and *Global_Redistribution*, then the insertion and deletion algorithms performance is not affected. So the crucial issue is to study the redistribution algorithms performance. The idea is that there will be a lower number of exceptions in r_e than the number of tuples in r_1, so the redistribution will be fast.

The worst case appears when we insert pairs of tuples (t_i, t_{i+1}) with the same X-value but

different Y-value, i.e:

$$t_i[X] = t_{i+1}[X] \neq t_j[X] \ (j > i+1) \quad , \quad t_i[Y] \neq t_{i+1}[Y]$$

In this case, the redistribution algorithms have $O(\#(r_e))$. If we take into account the insertion and deletion tasks, the redistribution algorithms have $O(\#(r_e) \log \#(r_e))$. Anyway, considering the worst case is not realistic because when the database is big enough, there will be almost no redistributions, because the exceptions will be clearly identified. Moreover, redistributions are usually performed at the beginning of the database life, and in such situation, there are only a few tuples (resulting a small value for $O(\#(r_e) \log \#(r_e))$). Therefore, we can conclude that efficiency is not affected, specially when the database is not *too young*.

4.5 Integrity and Security

In this section we are going to mention some important aspects the database administrator should take into account when working with relations derived from a partial decomposition.

The algorithms presented in the previous sections have the objective to manage all the updates performed over the partial decomposition. But they cannot guarantee that the semantics of the reality being modelled is going to be preserved.

The reason behind decomposing a given relation through a p.f.d is that such a relation almost satisfies a functional dependency: the system should be able to accept *exceptions*, but they should be treated in a special way. In fact, a major problem appears when an erroneous tuple is inserted into the database. Such a tuple would be accepted and stored into the exception relation. This is the price we have to pay for softening the restriction imposed by a functional dependency. The only way to solve this problem is to have a good enough problem domain knowledge and exploit it by using the RDBMS capabilities. For example, using attribute value restrictions, or controlling somehow the exceptions by the definition of triggers which will be fired when an update is performed on r_e.

Regarding the security issue, this can help to the database administrator to enforce the semantics of the p.f.d. Let us use an example to illustrate it. Let us consider a car dealership where the car price depends on the car model and its extras. We opt for a p.f.d instead of a f.d because some privileged clients (not too many) could obtain better prices than standard ones. It makes sense thinking that such discounts cannot be applied by all the sellers, only by some employees with special privileges such as the dealership director. In other words, these people should be granted by the database administrator to be able to update the exception relation, while the rest of the sellers should not have that privilege. It is worth mentioning that the director could insert an erroneous tuple into r_e, and, in this case, previous paragraph discussion applies. So, it is clear that we can establish a soft restriction and prevent from errors updating r_e through appropriately establishing the access permissions.

5 Conclusions

In this work we have introduced a mechanism to decompose a relation which *almost* verifies a functional dependency. We have also established the *MNE* restriction which assures the decomposition integrity when updates are performed over the decomposed scheme, and we have introduced several algorithms for managing these updates. It is not necessary to have a RDBMS supporting partial dependencies specifically. The RDBMS should incorporate stored procedures and exception management (in the programming sense), however. Our goal is achievable due to the universal relation concept which hides the internal decomposition to the user.

Some problems remain unsolved and will be treated in future works, such as studying the discovery of *good* partial dependencies in relations with actual data, raising the translation of p.f.d to data warehousing systems, deepening in the security and integrity issues, as well as establishing a general database normalization process for p.f.d.

References

[1] A. Borgida. Language features for flexible handling of exceptions in information systems. *ACM Transactions on Database Systems*, 10(4):565–603, December 1985.

[2] P. De Bra. *Horizontal Decompositions in the Relational Database Model.* PhD thesis, Antwerpen University, 1987.

[3] P. De Bra and J. Paredaens. Horizontal decompositions for handling exceptions to functional dependencies. *Advances in Database Theory*, II:123–144, 1983.

[4] M.A. Casanova, R. Fagin, and C.H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28:1:., .

[5] E.F. Codd. Normalized data base structure: A brief tutorial. In *Proceedings of ACM SIGFIDET Workshop on Data Description, Access, and Control*, San Diego, 1971.

[6] E.F. Codd. Further normalization of the data base relational model. In *Courant Computer Science Symposia*, volume 6 of *Englewood Cliffs*. Prentice Hall, N.J, 1972.

[7] E.F. Codd. Recent investigations in relational database systems. In *IFIP 74 Conf.*, pages 1017–1021. North Holland, 1974.

[8] S.S. Cosmadakis, P.C. Kanellakis, and N. Spyratos. Partition semantics for relations. In *Proc. Of the* 4^{th} *Symposium on Principles of Database Systems*, pages 261–275. ACM, 1985.

[9] R. Fagin. Multivaluated dependencies and a new normal form for relational databases. *ACM TODS*, 2:3:262–278, September 1977.

[10] I.J. Heath. Unacceptable file operations in a relational database. In *ACM SIGFIDET Workshop on Data Description, Access, and Control*, San Diego, 1971.

[11] D. MaierI. *The Theory Of Relational Databases.* Computer Science Press, 1803, Research Blvd. Rockville, Maryland 20850, 1983.

[12] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[13] J. Rissanen. Independent components of relations. *ACM TODS*, 2:3:317–325, December 1977.

[14] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.

[15] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II: The new technologies, chapter The Universal Relation as a User Interface, pages 1026–1069. Computer Science Press, 1988.