Updating OLAP Dimensions

Carlos A. Hurtado — University of Toronto — chl@db.toronto.edu
Alberto O. Mendelzon — University of Toronto — mendel@db.toronto.edu
Alejandro A. Vaisman — Universidad de Buenos Aires — av2n@dc.uba.ar

July 6, 1999

Abstract

OLAP systems support data analysis through a multidimensional data model, according to which data facts are viewed as points in a space of application-related "dimensions", organized into levels which conform a hierarchy. Although the usual assumption is that these points reflect the dynamic aspect of the data warehouse while dimensions are relatively static, in practice it turns out that dimension updates are often necessary to adapt the multidimensional database to changing requirements. These updates (although having received little attention in the research literature) can take place either at the structural level (v.g. addition of categories or modification of the hierarchical structure) or at the instance level(elements can be inserted, deleted, merged, etc.), and are poorly supported (or not supported at all) in current commercial systems. In a former paper [6] we introduced a formal model supporting dimension updates. Here, we extend the model, adding a set of semantically meaningful operators, which allow to perform some usual dimension updates in a more efficient way. We also formally define the mapping from the multidimensional to the relational model, and study how an implementation of the dimension updates could proceed, in a ROLAP storage of the data warehouse.

1 Introduction

The term OLAP (On Line Analytical Processing) refers to data analysis over large collections of historical data(*data warehouses*), in order to support the decision-making process, allowing the analyst to perform analysis of factual data(*e.g.* daily sales in the different branches of a supermarket chain) according to dimensions of interest(*e.g.* regions, products, stores, etc.). Data is often stored in OLAP servers, and client tools are provided by vendors, in order to facilitate visualization. These servers can be either relational databases(ROLAP) or multidimensional databases, which store data in proprietary multidimensional arrays (MOLAP). Some vendors are beginning to provide hybrid systems, which they call HOLAP [13].

Several works have proposed formal models for OLAP applications [1, 5, 4, 3]. Most of these are based on the original idea of the "star schema," [8] in which data is stored in a set of *dimension tables* and *fact tables*. The usual assumption here is that data in the fact tables reflect the dynamic aspect of the data warehouse, whereas data in the dimension tables represent basically static information about the "dimensions" according to which factual data will be analyzed. However, in practice, the evolution of the information stored in the warehouse requires the update of some of the dimensions. For instance, if we think of the data warehouse as a materialized view of data located in multiple sources [11, 15], we can imagine a situation in which the structure of these sources changes, or a new source is added, or an old one dropped. Any of these changes may require updates to the structure of some dimensions. Further, as multidimensional views are designed according to requirements from end users (a point highly emphasized in many industrial white papers [7, 12]), a redefinition of the initial requirements may cause a dimension update.

In previous work [6] we introduced a multidimensional model that includes a framework for dimension updates and a set of primitive dimension update operators. Here, we add to the primitive operators a set of semantically meaningful complex operators that encapsulate common sequences of primitive operators. These complex operators

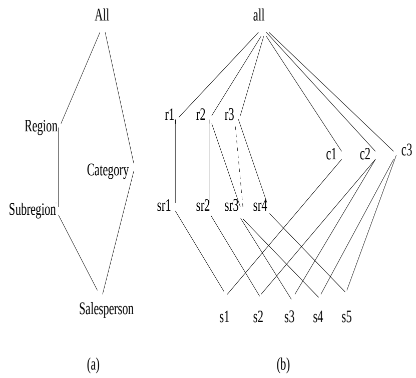

Figure 1: (a)Dimension Schema (b)Reclassifying regions.

can be implemented more efficiently than the equivalent sequences of primitive operators. We also sketch algorithms for implementing both the primitive and the complex operators in a ROLAP star schema environment. We compare two different approaches, normalized and denormalized dimension tables, from the point of view of dimension update algorithms.

Let us introduce the problem by means of a motivating example. Consider a star schema with a dimension representing salespersons, grouped into sub-regions and regions, and also grouped into categories (we do not care how these categories are defined). The dimension hierarchy is depicted in figure 1. Suppose, due to a business decision, we want to reclassify sub-region sr_3 to region r_3. Applying the primitive operators we defined [6], we need to delete s_3, s_4, sr_3, and sr_4, and then resinsert them with the appropriate relationships among them:

$DelInstance$(d, salespersons, s_3),
$DelInstance$(d, salespersons, s_4),
$DelInstance$ (d, SubRegion, sr_3),
$AddInstance$ (d, SubRegion, sr_3,$\{r_3\}$),
$AddInstance$ (d, salespersons, s_3,$\{sr_3, c_2\}$),
$AddInstance$ (d, salespersons, s_4,$\{sr_3, c_3\}$).

In a denormalized relational implementation, as we will see, these operations would imply that we delete all tuples t s.t. $t.salespersons = s_3$, and $t.salespersons = s_4$, inserting them afterwards with a value $t.region = r_3$. A more difficult problem would arise if we wanted to re-organize the classification hierarchy, merging, for instance, sub-regions sr_2 and sr_3 into a new one sr_n, a not unusual commercial decision. We would obtain a dimension like the one depicted in figure 2. Again, using the basic set of operators, we would need to perform the following sequence:

$DelInstance$ (d, salespersons, s_2),
$DelInstance$ (d, salespersons, s_3),
$DelInstance$ (d, salespersons, s_4),
$DelInstance$ (d,SubRegion,sr_2),
$DelInstance$ (d, SubRegion, sr_3),
$AddInstance$ (d, SubRegion, sr_n,$\{r_2\}$),
$AddInstance$ (d, salespersons, s_2,$\{sr_n, c_2\}$),
$AddInstance$ (d, salespersons, s_3,$\{sr_n, c_2\}$),
$AddInstance$ (d, salespersons, s_4,$\{sr_n, c_3\}$).

Finally, we may want to split a region into two or more, perhaps because it has become too crowded and the sales force is not enough to attend to it. Again, this would require a sequence of applications of the primitive operators.

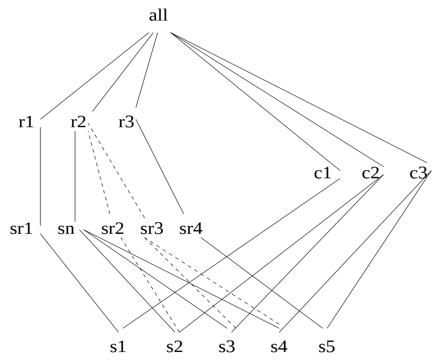

Figure 2: Merging sub-regions.

In this paper we will introduce more powerful operators, equivalent to sequences of primitive operators like those shown in the example, along with a mapping of the model and the update operators to the relational model. The contributions of this paper are :

- An extension of the multidimensional model formerly introduced [6], with a set of *complex operators* which can be implemented more efficiently than sequences of primitive operators.
- A mapping from the multidimensional to the relational model, using two different approaches: normalized and denormalized dimension tables, and some properties of these mappings.
- A discussion of the algorithms that implement the primitive as well as the complex operators, both in the normalized and denormalized implementations.

The paper is organized as follows : In Section 2 we review the multidimensional model supporting dimension updates. In section 3 we introduce the new update operators. In section 4 we map these operators to the relational model, and present some properties of this mapping. In Section 5 we present the algorithms for the update operators, concluding in Section 6.

2 Dimension Updates

In this section, we give an overview of the data model which was introduced in a former work [6]. We refer the interested reader to that work, where he/she will find a complete description of the model. We also introduce new operators and define some of their properties. We define properties which will allow us to define whether an element should be an attribute or a level of a dimension, along with a discussion on the model.

2.1 Modeling Dimensions

Assume the following sets: a set of level names $\mathbf{L}$, where each level $l \in \mathbf{L}$ is associated with a set of values $dom(l)$;a set of attribute names $\mathbf{A}$ with an associated domain $dom(a)$; a set of dimension names $\mathbf{D}$; and a set of fact table names $\mathbf{F}$.

We start by defining the notion of *dimension schema* which is a DAG(Directed Acyclic Graph), representing the multiple hierarchy of levels of the dimension.

Definition 1 (Dimension Schema) *A* dimension schema *is a tuple* (dname,$L, A, \preceq, \gg$)*, where dname* $\in \mathbf{D}$ *is the name of the dimension,* $L \subseteq \mathbf{L}$ *is a finite set of levels, which contains a distinguished level name All, such that* $dom(All) = \{all\}$*,* $\preceq$ *is a relation over levels, such that* $\preceq^*$*, its transitive and reflexive closure , is a partial order, with a unique bottom level, called* l_{inf}*, a unique top level, All, and, for every level* $l \in L$*,* $l_{inf} \preceq^* l$ *and* $l \preceq^* All$ *hold. Moreover, if* l_a *and* l_b *are levels in* L*, and* $l_a \preceq^* l_b$*, then* $l_a \not\preceq l_b$*. Finally,* $A \subseteq \mathbf{A}$ *is a finite set of attributes, and* $\gg$ *is a function from attribute names to level names, s.t.* $a \gg l$ *means that* $\gg$ *applied to an attribute* a *returns the level* l *the attribute belongs to.*

An instance of a dimension is obtained by specifying a set of *roll-up functions*, one for each edge in the dimension schema. An important feature of the model is that it is allowed to represent some sort of incomplete dimensions: dimensions where some levels do not have associated elements. For instance, we might want to add a brand but do not yet have products that belong to it.

Definition 2 (Dimension Instance) *A* dimension instance *is a tuple* $(D, RUP, DESC)$*, where* D *is a dimension schema,* RUP *is a set of partial functions such that :*

1. *for each pair of levels* l_1*,* l_2 *such that* $l_1 \preceq l_2$*, there exists a roll-up function* $RUP_{l_1}^{l_2} : dom(l_1) \rightarrow dom(l_2)$ *in* RUP*.*
2. *for each pair of paths, in the graph with nodes in* L *and edges in* $\preceq$*,* $\tau_1 =< l_1, l_2, \ldots, l_{n-1}, l_n >$*, and* $\tau_2 =< l_1, l'_2, \ldots, l'_{m-1}, l_m >$*,*$l_n = l_m$*, we have* $RUP_{l_1}^{l_2} \circ \ldots \circ RUP_{l_{n-1}}^{l_n} = RUP_{l_1}^{l'_2} \circ \ldots \circ RUP_{l'_{m-1}}^{l_m}$*.*
3. *for each triple of levels* $l_1, l_2, l_3 \in L$ *such that* $l_1 \preceq l_2$ *and* $l_2 \preceq l_3$*,* $ran(RUP_{l_1}^{l_2}) \subseteq dom(RUP_{l_2}^{l_3})$*.*
4. $DESC$ *is a partial function with signature* $DESC_l^a : dom(l) \rightarrow dom(a)$ *, for each level* l *and attribute* a*, such that* $a \gg l$*.*

In what follows, we will refer to the second condition of the former definition as the *consistency* condition. Thus, an instance verifying this condition will be denoted as *consistent.*

Not any set of roll up functions are consistent to allow summarizability. Consider the following example [14]: assume we have a set of classes which classify physicians by specialty, and we would like to give the number of physicians in each specialty, in order to produce a summary dataset; this summarization will produce erroneous results, since physicians with multiple specialties will be counted multiple times. Another example from the same source reflects another possible anomalies in summarization: if we are given populations of all cities whithin a state, it is clearly impossible to summarize that to the state level (to mean the population of a state), because of the missing villages, farms, etc.; in this case, the problem is that classification is incomplete relative to the measure *population*. The *summarizability property* [14] consists of a set of constraints imposed over the dimension instance to avoid such anomalies. The following constraint has been proposed to handle this problem [9] : a level of a dimension is summarizable if it contains elements that form a partition of the set of elements in the bottom level of such dimension.

In our model, levels are summarizable, while attributes might not be. The summarizability property is imposed by the following constraints presented in Definition 2:

- The range of every roll-up function is a subset of the domain of the roll-up function above it. This means that any value in the lower level is mapped to some value in the upper level, thus a measure associated to it is not lost in the summary.
- Given two paths in the dimension schema starting and ending at the same level, the composition of the roll-up functions must be the same. This ensures that in the common final level, each element belongs to a single class.

Example 1 *Figure 3 shows an example of a dimension schema and instance for a dimension* Product*. Here, for the dimension schema, dname = Product, L = {ItemId, Brand, Company, Category, Corporation, Product, All}.* $A = \{UPC, Desc, Price\}$*. Relation "*$\preceq$*" contains the following pairs: ItemId* $\preceq$ *Brand, Brand* $\preceq$ *Company, Company* $\preceq$ *Corporation, Corporation* $\preceq$ *All, ItemId* $\preceq$ *Category, and* $UPC \gg$ *Item,* $Desc \gg$ *Item,* $Price \gg$ *Item. For the sake of clarity, do not show the attributes in this figure, and in the subsequent ones.*

In what follows, dimension will stand for dimension instance, except when noted. We will also denote by RUP^* the set of roll up functions that indirectly relate the instances of the levels (RUP denotes the direct rollup functions); this set contains a roll up function for each pair of levels $l_m, l_n \in L$, $l_m \preceq^* l_n$, such that if $l_m = l_n$, $RUP_{l_m}^{l_n} =$ *identity*; otherwise, if $< l_m \ldots l_n >$ is a path from l_m to l_n in the graph with nodes in L and edges in $\preceq$, $RUP_{l_m}^{l_n} = RUP_{l_m}^{l_{m+1}} \circ \ldots \circ RUP_{l_{n-1}}^{l_n}$.

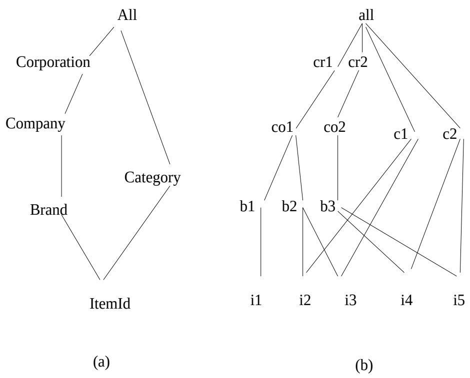

Figure 3: (a) A dimension schema for Product, (b) A dimension instance for Product.

It can be shown that roll up functions defined over the same level must have the same domain, the set of all values at that level. More formally, given a dimension instance, for each triple of levels l, l' and l'' of it, such that $l \preceq l'$ and $l \preceq l''$, $dom(RUP_l^{l'}) = dom(RUP_l^{l''})$.

Given a dimension and a pair of levels l and l', such that $l \preceq l'$, the **instance set** of l, or $instset(l)$, is the set containing the values in l. Analogously, if e is an attribute, $instset(e) = ran(DESC_l^e)$, where l is a level such that $e \gg l$.

It may appear, at first sight, that attributes and levels could be freely interchanged, making the distinction superfluous. However, there may be cases in which some elements $i \in instset(l_i)$ have no rollup defined for them to another level l_j, s.t. $l_i \preceq l_j$. Then, we could not define l_j as a dimension level, because, among other reasons, the *summarizability property* would be violated.

Now, we will present here a summary of the basic set of operators which we introduced, as we have already mentioned, in a former work [6] . Operators dealing with attributes, are straightforward and we omit them.

2.2 Structural Update Operators

Operator 1 (Generalize) *This operator creates a new level, call it l_n, in a dimension d, to which a pre-existent one, l, will roll up. The new level cannot exist in d. A function f must be defined from the instance set of l, to the domain of l_n, preserving the dimension's consistency. Thus, Generalize is expressed as* Generalize$(d, l, l_n, f_l^{l_n})$.

Example 2 *Let us suppose a dimension* Store, *with levels* Region *and* All. *For design reasons which we are not going to analyze at this point, we want to define a new level in it, say, the type of the store, call it* Type, *with instant set $instset(Type) = \{t_1, t_2\}$. The operation would be defined as* Generalize(Store, StoreId, Type, $f_{StoreId}^{Type}$), *where $f_{StoreId}^{Type} = \{(s_1, t_1), (s_2, t_1), (s_3, t_2)\}$.*

Operator 2 (Specialize) *This operator adds a new level l_n to a dimension d, such that l_n will roll up to the lowest level of d, l_{inf}. Thus, it will become the lowest level of the new dimension. Again, a function f must be defined for this roll up. The operator is expressed as* Specialize$(d, l_n, f_{l_n}^{l_{inf}})$

Example 3 *Suppose now that we want to specialize the dimension* Time, *where* day $\preceq$ week $\preceq$ All. *The only level at which we can do this is* Day. *The operator will be defined as :* Specialize(Time,Hour,f_{day}^{Hour}).

Operator 3 (Relate Levels) *This operator defines a roll up function between two independent levels l_a and l_b, in a dimension d. A necessary condition for this, is the existence of a function f between the instance sets of the levels being related, such that the dimension instance remains consistent. If this is not accomplished, the levels cannot be related. Moreover, when conditions for relating l_a and l_b are met, we must delete all the redundant roll up functions that may appear, which are not admitted in our model(remember we only include direct roll ups in it). For instance, we must delete the roll up functions between the levels l and l_b such that $l \preceq^* l_a$ and $l \preceq l_b$. Thus, the operator is expressed as:* Relate(d, l_a, l_b). *Note that f is implicitly defined.*

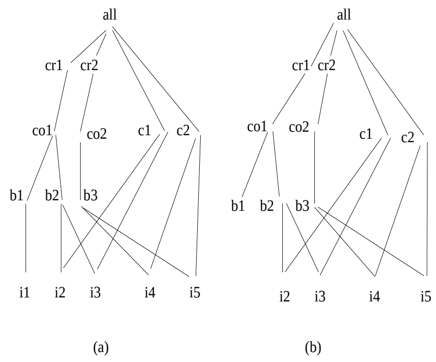

Figure 4: (a)Add instance (b) Delete instance.

Example 4 *We could relate level* Brand *with level* Category *in the dimension* Product. *Notice that we have deleted the relation* Item $\preceq$ Category *as a colateral action. The operator will have the form :* Relate(Product, Brand, Category).

Operator 4 (Unrelate) *This operator deletes a relation $\preceq$ between two levels l_a and l_b, such that $l_a \preceq l_b$, in a dimension d . The operator must guarantee that levels which are below than l_a in the hierarchy will still be able to reach the same levels they reached before the unrelate operation. For instance, if $l_a \preceq l_b$ and $l_b \preceq l_c$, we must preserve $l_a \preceq l_c$ by making it explicit, in case we delete $l_a \preceq l_b$, because, again, this relation was only implicit in the model. Furthermore, we must connect l_a with every level l_k such that $l_b \preceq l_k$. Then, we have :* Unrelate(d, l_a, l_b).

Operator 5 (Delete Level) *This is an operator that deletes a level l and its roll up functions, from a dimension d. The level to be deleted cannot be the l_{inf}, unless it rolls up to only one higher level. As it was the case of* Relate *operator, taking into account that we only define the direct roll ups, when deleting a level, we must add the functions among levels above and below the level being deleted. The operator is invoked as* DelLevel(d, l).

2.3 Instance Update Operators

The following operators add or delete instances belonging to a level in a dimension . They also impose some constraints to the way these updates can be performed.

Operator 6 (Add Instance) *This operator inserts a new element, say x, into a level l_a(i.e., an element not belonging to the instance set of l_a), in a dimension d. We must provide the operator with a set P of pairs (l_i, x_i), such that every l_i is a level to which l_a rolls up($l_a \preceq l_i$), and $RUP_{l_a}^{l_i}(x) = x_i$(in the updated dimension). We invoke the operator as :* AddInstance(d, l_a, x, P).

Operator 7 (Delete Instance) *This operator deletes an element x_a, belonging to the instance set of a level l_a, in a dimension d. It is only defined when no element of any level l_i such that $l_i \preceq l_a$, rolls up to x_a. Then, we have* DelInstance(d, l_a, x_a).

Example 5 *Suppose we want to add a new item, say i_5, to an instance of the* Product *dimension, by means of :* AddInstance (Product ,$ItemId, i_5, \{(Brand, b_3), Category, c_2)\}$). *Figure 4(a) shows the result. After that, we delete i_1 from level* ItemId. *Note that we can only, in this case, delete an element belonging to* instset(ItemId), *due to the operator preconditions. The operation is invoked as* DelInstance(Product, ItemId, i_1). *See figure 4(b).*

3 Complex operators

Many common changes to dimensions result in long sequences of primitive updates using only the operators defined so far. In this section we introduce *complex operators*, intended to capture such common sequences and encapsulate them in a single operation.

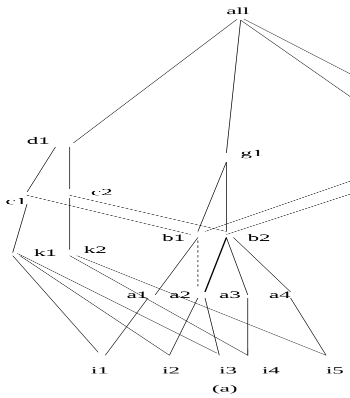

Figure 5: Reclassification

The set of complex operators, all of them included in which we called as *instance updates* because they affect the instances of a dimension, is composed by the following:

- Reclassify
- Split
- Merge
- Update

We will treat each one separately.

3.1 Reclassify.

Suppose a brand is purchased by a new company, or different regions assigned to salespersons as a result of marketing decisions, stores assigned to different regions, etc. All of these operations could be performed as a transaction involving a series of $DelInstance$ and $AddInstance$ operations. We must note, however, that a single one, call it $Reclassify$, can do the job in a single step. However, this reclassification may or may not lead to a consistent dimension. Thus, we must give the conditions under which the operator could be defined.

Operator 8 (Reclassify operator) *Given a dimension* d*=*$((dname, L, A, \preceq, \gg), RUP, DESC)$*, a pair of levels* l_a *and* l_b*, a pair of elements* $x_a \in instset(l_a)$ *and* $x_b \in instset(l_b)$; $\mathrm{Reclassify(d, l_a, l_b, x_a, x_b)}$ *is a new dimension* $((dname, L, A, \preceq, \gg), RUP', DESC)$ *s.t.:*

- $RUP'^{l_b}_{l_a} = RUP^{l_b}_{l_a} \setminus \{(x_a, x_j) | RUP^{l_b}_{l_a}(x_a) = x_j\} \cup \{(x_a, x_b)\}$;
- $RUP'^{l_j}_{l_i} = RUP^{l_j}_{l_i}$*, for all other levels* l_i, l_j*.*
- *The new dimension remains consistent, i.e., verifies all the properties stated in definition 2.*

Reclassify is not defined for every possible dimension, as is shown in figure 5 . Here, if we reclassify a_2 from b_1 to b_2, consistency would be violated. However, note that as no level rolls up to level E, no problem arise form it, but from level C, which is reached from l_b and from l_{inf}. This observation leads to a general condition under which reclassification may always be performed.

Definition 3 (Conflicting Levels) *Let us suppose we have* $d, l_a, l_b, l_a \preceq l_b, x_a, x_b$*, defined as in operator 8. We say a level* $l_k \in d$ *s.t.* $l_b \preceq^* l_k$*, is* conflicting *w.r.t. reclassification, if there exists a level* l_i *s.t.* $l_i \preceq^* l_a$*, there is an alternative path between* l_i *and* l_k *not including* (l_a, l_b)*, and* x_a *is reached by at least one element in* $instset(l_i)$*.*

Theorem 1 (Definiteness of the Reclassify Operator) $\mathrm{Reclassify(d, l_a, l_b, x_a, x_b)}$ *is defined if and only if for every conflicting level* l_k*,* $RUP*^{l_k}_{l_b}(x) = RUP*^{l_k}_{l_b}(x_b)$ *holds, where* $RUP^{l_b}_{l_a}(x_a) = x$*.*

Proof(Sketch). We will first prove the "only if" part of the theorem. Let us suppose that l_k is a *conflicting* level and there exists an element $e \in instset(l_i)$ (l_i defined as above) s.t. e reaches x_a. Moreover, let us suppose that $RUP*_{l_b}^{l_k}(x_b) = e_1, RUP*_{l_b}^{l_k}(x) = e_2, e_1 \neq e_2$. In this case, it is clear that *Reclassify*(d, l_a, l_b, x_a, x_b) would lead to an inconsistent graph, because $RUP*_{l_i}^{l_k}(e) = e_1$, and $RUP*_{l_i}^{l_k}(e) = e_2$, through two different paths.
The "if" part of the theorem proceeds as follows. We assume that *Reclassify*(d, l_a, l_b, x_a, x_b) is defined, thus, it must lead to a consistent dimension. Thus, either of the following must hold : (1)no conflicting level exist, in which case, nothing else is required, and the reclassification only implies shifting the rollup function $RUP_{l_a}^{l_b}(x_a)$ from x to x_b, or (2) l_k is a conflicting level w.r.t the proposed reclassification. In this case, let us suppose, again, that $e \in instset(l_i)$ exists, and $RUP*_{l_i}^{l_a}(e) = x_a$. As l_k is conflicting, if $RUP*_{l_i}^{l_k}(e) = e_1$ through a path not including (l_a, l_b), as, by hypothesis, reclassify is defined, this must also hold through any path, including one passing through l_a, l_b or both. Thus, $RUP*_{l_b}^{l_k}(x) = RUP*_{l_b}^{l_k}(x_b)$ must hold.

3.2 Split.

Suppose for instance, some country is divided into four regions, *north, south, east, west*, in order to assign salespersons, and someone decides that the northern region should be divided into two or more, because it is getting too crowded, and more sales reps must be assigned to it. We need an operator that can handle this situation, this is, a operator which lets the user specify which salesman is assigned to each region, and, automatically reorganize the dimension, keeping its consistency. Formally :

Operator 9 (Split operator) *Given a dimension* d=$((dname, L, A, \preceq, \gg), RUP, DESC)$*, a level* l_a*, an element* $x_a \in instset(l_a)$*, a list* E *of the form* $\{x_{a1}, \ldots x_{an}\}$*, where* $x_{ai} \in dom(l_a) \setminus instset(l_a)$*, another list* P *of the form* $P = \{x_{a1}[(l_1 : list_1) \ldots (l_m : list_m)]; \ldots; x_{an}[(l_1 : list_1) \ldots (l_m : list_m)]\}$*, where* $l_i \preceq l_a, i = 1..m$*,* $\mathrm{list_i}$ *is a list of elements in* $\mathrm{instset(l_i)}$*, of the form* $(x_1, \ldots, x_k)$ *s.t.* $RUP_{l_i}^{l_a}(x_i) = x_a$*;* $\mathrm{Split(d, l_a, x_a, E, P)}$ *is a new dimension* $((dname, L, A, \preceq, \gg), RUP', DESC)$*,where :*

- $RUP'^{l_a}_{l_i} = RUP^{l_a}_{l_i} \setminus \{(x_i, x_a) | RUP^{l_a}_{l_i}(x_i) = x_a \cup \{(x_i, x_{aj}) | x_{aj} \in E, x_i \in list_i \ s.t.\ x_{aj}[l_i : list_i] \in P\}$;
- $RUP'^{l_i}_{l_a} = RUP^{l_i}_{l_a} \setminus \{(x_a, x_i) | RUP^{l_i}_{l_a}(x_a) = x_i\} \cup \{(x_{aj}, x_i) | x_{aj}[l_i : list_i] \in P, x_{aj} \in E \wedge RUP^{l_i}_{l_a}(x_a) = x_i\}$
- $RUP'^{l_j}_{l_i} = RUP^{l_j}_{l_i}$*, for all other levels* l_i, l_j*.*
- *The new dimension must remain consistent.*

Example 6 *Suppose we want to split element* d *in figure 6, into two elements* $\mathrm{d_1}$ *and* $\mathrm{d_2}$ *in the domain of level* D*. The operation will be denoted as :* $\mathrm{Split}(dname, D, d, \{d_1, d_2\}\{d_1 : (B : (b_1, b_2)), (C : (c_1, c_2)); d_2 : (B : (b_3)), (C : (c_3))\})$*. This expression assigns a set of elements in every level reaching directly level D(i.e. B and C), to each new element into which d splits (i.e.* d_1 *and* d_2*). Note that the user must assign the roll up functions corresponding to the new values* $\mathrm{d_1}$ *and* $\mathrm{d_2}$*, and this assignment must be s.t. the dimension remains consistent. In this example,* b_1, b_2, c_1 *and* c_2*, were assigned to* d_1*.*

3.3 Merge.

The *Merge* operator performs the inverse of *Split*, i.e., it merges two instances of a dimension into a single one. For instance, several airlines could become a single one as a result of a corporate fusion.

Operator 10 (Merge operator) *Given a dimension* d=$((dname, L, \preceq), RUP, DESC)$*, a level* l_a*, an element* $x_N s.t. x_N \in dom(l_a) \wedge x_N \notin instset(l_a)$*, a set of elements* $X = \{x_1 \ldots x_n\} \in instset(l_a)$ *s.t. all the elements* $x_i \in X$ *roll up to the same element in every level* l *s.t.* $l_a \preceq l$*;* $\mathrm{Merge}(d, l_a, X, x_N)$ *is a new dimension* $((dname, L, \preceq), RUP', DESC)$*,where :*

- $RUP'^{l_a}_{l_i} = RUP^{l_a}_{l_i} \setminus \{(x_i, x_j) | RUP^{l_a}_{l_i}(x_i) = x_j, x_j \in X\} \cup \{(x_i, x_N) | RUP^{l_a}_{l_i}(x_i) = x_j, x_j \in X\}$;
- $RUP'^{l_j}_{l_a} = RUP^{l_j}_{l_i} \setminus \{(x_i, x_j)\} \cup \{(x_N, x_j)\}, x_i \in X, RUP^{l_j}_{l_a}(x_i) = x_j, l_a \preceq l_j$;

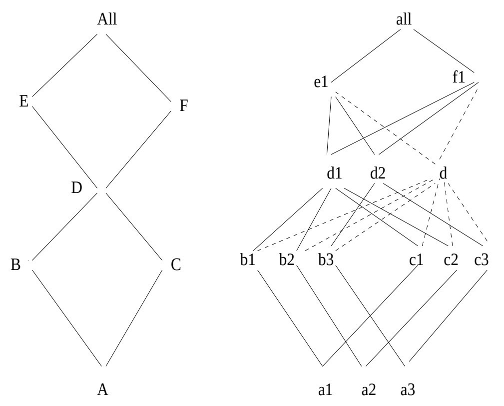

Figure 6: Split operator.

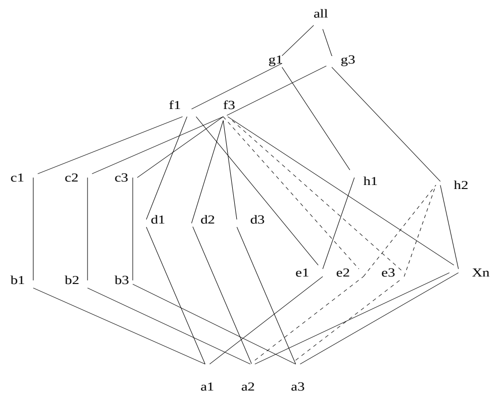

Figure 7: Merge operator

- $RUP'^{l_j}_{l_i} = RUP^{l_j}_{l_i}$, *for all other levels* l_i, l_j.
- *The new dimension remains consistent, i.e., verifies all the properties stated in definition 2.*

Example 7 *Figure 7 shows an example of the above definition. We can see that the operation* $\mathrm{Merge(d, E, \{e_2, e_3\}, X_n)}$ *keeps the dimension in a consistent state.*

Note that we may have omitted the condition stating that every element in X must roll up to a single element in the levels above l_a in the hierarchy. However, this condition makes *Merge* and *Split* symmetric operators, this is, $Merge(Split(d)) = d$. For instance, in example of figure 6, if after the split we subsequently perform a merge of the form $Merge(dname, D, \{d_1, d_2\}, d$, this operation will turn the dimension back to its original configuration.

3.4 Update.

We also need an operator such that only changes the value of an element, keeping the structure and rollup functions unchanged. For instance, suppose a brand name for a set of items changes, say, for marketing reasons, but the corporation for that brand remains the same, as well as the set of products related to the brand. Again, all these changes could be reflecetd by a sequence of individual insertions and deletions.

Operator 11 (Update operator) *Given a dimension d=*$((dname, L, A, \preceq, \gg), RUP, DESC)$*, a level* l_a*, an element* $x_a \in instset(l_a)$*, and an element* $x_n \notin instset(l_a)$; $\mathrm{Update(d, l_a, x_a, x_n)}$ *is a new dimension* $((dname, L, A, \preceq, \gg), RUP', DESC)$ *s.t.:*

- $RUP'^{l_j}_{l_a} = RUP^{l_j}_{l_a} \setminus \{(x_a, x_j) | RUP^{l_j}_{l_a}(x_a) = x_j\} \cup \{(x_n, x_j) | RUP^{l_j}_{l_a}(x_a) = x_j\}$;

- $RUP'^{l_a}_{l_j} = RUP^{l_a}_{l_j} \setminus \{(x_j, x_a) | RUP^{l_a}_{l_j}(x_j) = x_a\} \cup \{(x_j, x_n) | RUP^{l_a}_{l_j}(x_j) = x_a\}\}$;
- $RUP'^{l_j}_{l_i} = RUP^{l_j}_{l_i}$, *for all other levels* l_i, l_j.

3.5 Efficiency issues

We claimed in section 1 that the complex operators introduced here, perform better than an equivalent sequence of primitive ones. We will show the truthfulness of this assertion making use of a comparison between the number of updates required in each case, to be performed over the graph representing the instance of the dimension . Thus, we assume that this metric takes into account the costs of checking the operators' preconditions. We will make use of the example introduced in section 1.
Reclassify. In the dimension depicted in figure 1 a reclassification of subregion sr_3 from region r_2 to region r_3 is shown. As was already described in section 1, due to the preconditions of the basic operators, six of such operations would be necessary in order to perform the former dimension update. Now, we will analyze how many graph operations would this sequence require. *DelInstance*(d, salespersons, s_3), and *DelInstance*(d, salespersons, s_4) take two edge deletions each one over the graph representing the instance of the dimension. *DelInstance* (d, subregion, sr_3) takes an additional edge deletion. *AddInstance* (d, subregion, sr_3,$\{r_3\}$) takes one edge addition, and finally *AddInstance* (d, salespersons, s_3,$\{sr_3, c_2\}$) and *AddInstance* (d, salespersons, s_4,$\{sr_3, c_3\}$) take two edge additions each. This gives a total of ten graph operations over the dimension *Salespersons*. On the other hand, applying the reclassification operator, defined for this case as *Reclassify*(*Salespersons*, *subregion*, *region*, sr_3, r_3), involves only one edge deletion and one edge addition, giving a total of two.
Although this is a particular case, as it was expressed in the proof of theorem 1, when the preconditions for reclassification hold, it only takes a shift on the corresponding rollup function to perform it, which takes just an edge deletion and an edge addition.
Merge. We proceed in a similar fashion for the *Merge* operator. In this example, the sequence described in section 1 would take eight edge deletions over the graph, plus seven edge additions. On the contrary, taking advantage of the *Merge* operator defined as *Merge*(*Salespersons*, *subregion*, s_n, $\{sr_2, sr_3\}$) would take only three edge additions, after checking that the preconditions for the operator hold. However, unlike *Reclassify*, for the *Merge* operator, the amount of graph operations required depends on the number of levels which roll up to the one being merged, and on their cardinality.

For the *Split* and *Update* operators, the same concepts apply. For the sake of brevity we will not extend on them here.

4 Mapping Dimensions to Relations

In this section, we present a mapping from the multidimensional model to the relational model. We study two possible implementations: (a) dimensions are stored as a single table(denormalized case) , and (b) dimensions are stored as a set of normalized tables.

4.1 Denormalized Relational Representation

We will denote the relational representation of a dimension d as a pair $R_d = (S, R)$, where $S = (rname, A, \mathcal{F})$ is the schema of the relation with $rname$ as the name of the dimension, A as its set of attributes; and $\mathcal{F}$ as its set of functional dependencies satisfying $dom(\mathcal{F}) \cup ran(\mathcal{F}) \subseteq A$. R represents the set of tuples in the relation(its instance). We will omit, in what follows, the trivial functional dependencies.

A dimension $d = ((L, A, \preceq, \gg), RUP)$, as it is represented in our multidimensional model, is mapped to the relational representation $R_d = (S_d, T_d)$, called *dimension table*, in the following way:

Schema The schema $S_d = (rname, A, \mathcal{F})$ of the relation is the following:

- $rname$ is $dname$.
- A contains an attribute l for each level $l \in L$.

- $\mathcal{F}$ contains a f.d. $rname : l_a \rightarrow l_b$ for each pair of levels $l_a, l_b \in L$ such that $l_a \preceq l_b$.

Instance The set of tuples T_d in the relation is defined as follows:

- Let us define the leafs of a level $l \in L$, $Leafs(l)$, as the set of elements in $InstSet(l)$ which are not reached by any other element above in the dimension instance. Formally: $Leafs(l) = InstSet(l) \setminus (ran(RUP^l_{l_1}) \cup \ldots ran(RUP^l_{l_n}))$, where $l_1, \ldots, l_n$ are the levels directly below l in the hierarchy.
- For every level l, and for every element $e \in Leafs(l)$, we have a tuple t in T_d defined as follows:

$$t(l_i) = \begin{cases} RUP^{*l_i}_{l}(e) & \text{If } l \preceq^* l_i \\ null & \text{otherwise} \end{cases}$$

The fd's are only applied over non-null values, i.e., a null in l_a can be related to two different elements in l_b even if we have $l_a \rightarrow l_b$ in $\mathcal{F}$. This allows to have the denormalized relation with nulls in attributes associated to levels below leaf elements.

Example 8 *The denormalized representation of the dimension depicted in Figure* **??** *is the following:*

ItemId	Brand	Categ.	Comp.	Corp.	Price
i_1	b_1	c_1	co_1	cr_1	1500
i_2	b_2	c_1	co_1	cr_1	1000
i_3	b_3	c_2	co_2	cr_2	2000
i_4	b_3	c_2	co_2	cr_2	3000
..	..	..			

The following lemma shows that this mapping yields a minimal cover of the set of functional dependencies representing each dimension hierarchy.

Lemma 1 *Given a dimension d and its denormalized representation $R_d = ((rname, A, \mathcal{F}), T)$, (a) $\mathcal{F}$ is the only minimal cover for $\mathcal{F}$; and (b) T satisfies $\mathcal{F}$.*

4.2 Normalized Relational Representation

The mapping proposed in section 4.1 implies that the dimension tables will be denormalized, situation recommended by many authors [8]. However, it is worth to compare this approach to one in which dimension tables become normalized, in an environment in which dimension updates are supported. In this subsection section we propose a *normalized* mapping from the multidimensional model to the relational model, in order to perform the comparison between both approaches wrt dimension updates. The basic idea is to build a table for each functional dependency implied, as was already proved, by the dimension hierarchy.

The normalized relational representation of the dimension d, denoted $\mathcal{R}_d$, is a pair $(S_d, \mathcal{R}_d)$ where S_d is the schema and $\mathcal{R}_d$ the instance of the representation. $\mathcal{R}_d$ is defined as follows:

Schema $S_d = (rname, Sc, \mathcal{F})$ is defined as follows:

- $rname$ is $dname$
- For each level l_i such that $l_i \preceq All$ we have a relational schema $name(l_i, All)[l_i]$ in Sc ($name(l_i, All)$ denotes the name of the relation).
- For each pair of levels $l_i, l_j \in L$ such that $l_i \preceq l_j$ we have a relational schema $name(l_i l_j) : [l_i, l_j]$ in Sc
- Let r be the outer-join of the relations in Sc, for each pair of levels $l_i, l_j \in L$ such that $l_i \preceq l_j$ we have the functional dependency $r : l_i \rightarrow l_j$ in $\mathcal{F}$. As in the previous section, the fd's in $\mathcal{F}$ do not consider nulls.

Instance $\mathcal{R}$ is a set of relations defined as follows:

- For each relation schema $name(l_i, All)[l_i] \in Sc$ we have a relation with the elements in $InstSet(l_i)$ as tuples.

- For each relation schema $name(l_i, l_j)[l_i, l_j] \in Sc$ we have a relation with the pairs in $RUP_{l_i}^{l_j}$ as tuples.

In what follows we will only deal with dimension tables representing levels. The approach is extended to attributes of dimensions in a straightforward fashion. In addition, we will denote S_d the schema of the normalized representation, which is the set of attributes of all the relations, and a set of functional dependencies for the pairs of attributes in each relation.

5 Computing the Update Operators

In this section we present the main ideas about the algorithmic specification of the computation of each update operator in a relational representation. This will give a clue for their implementation using SQL. We present the details of the implementation of the *Relate*, *AddInstance*, and *Reclassify* operators. We refer the reader to the full version of this paper [] for details on the remaining operators.

5.1 Preliminaries

The schema of both, the normalized and denormalized representation $S_d = (A, \mathcal{F})$ can be seen as a Directed Acyclic Graph(DAG), where the vertices are the attributes in A, and the edges are the pairs in $\mathcal{F}$. As we will see below, the procedural implementation of the update operators requires the execution of a set of standard operations over this DAG. In the algorithms, we will use $Reach(l_a, l_b, S)$ to mean that there is a path from vertex l_a to vertex l_b in dag S, $ReachSet(l_a, S)$ to stand for the set of all vertices that can be reached indirectly from l_a, and $Preset(l_a, S)$ to stand for the set of all vertices that reach indirectly l_a.

In addition, we will use relational algebra operators augmented with the *Update* operator, which will allow us to express the update statement provided by SQL. The update operator is denoted as: $\mu(R, cond, t)$, where R is a relation, $cond$ is a condition of the relation, and t represents a tuple, with attributes in the set of attributes of R. The new relation results from updating these attributes of the tuples in R satisfying $cond$, with the values in t. We will use the following notation for the specification of the algorithms: $R_1, R_2, \ldots$, as variables for relations; $l_1, l_2, \ldots$ as variables for attributes; $A_1, A_2, \ldots$ as variables for sets of attributes; $S_1, S_2, \ldots$ as variables for dags; $t_1, t_2, \ldots$ as variables for tuples; $E_1, E_2, \ldots$ as variables for sets of values in the domain of some attribute; $F_1, F_2, \ldots$ as variables for relations which contain roll up functions. We will use $R(l)$ as a shorthand for representing a value for the attribute representing level l, which is the same for every tuple in R. for that value. We will also use sentences of the form $precondition$: $condition$ meaning that the algorithm halts with precondition false if $condition$ is false.

5.2 Structural Update Operators

The implementation of the structural update operators is straightforward. The algorithm performs four type of operations: (a) Operations that verify preconditions over the schema: for instance, when the *DelLevel* operator is applied, the algorithm checks that the resulting dimension has one bottom and one top level. Operations of this type have the same execution cost in both representations for all the operators. (b) Operations that verify preconditions over the instance: as an example, when the *Generalize* operator is applied, the algorithm checks that the domain of the added roll up function includes the instance set of the level being generalized. For the *Relate* operator, this type of operations have a higher cost in the normalized representation than the denormalized one, because the algorithm performs a set of joins between the relations in order to verify the added functional dependency. For the remaining operators, operations of type (b) have the same cost in both representations. (c) Operations that update the schema: operations which add/delete edges/levels to/from the schema, and compute the *transitive reduction* (i.e., the graph without transitive edges) [2] of the updated schema. They have the same cost for all the operators. (d) Operations that update the instance: in general, they are additions/deletions of roll up functions associated to edges added/deleted by means of the operations described in (c). Their execution cost is higher for the denormalized representation because when a roll up function is added the algorithm performs a join between the new roll up function and the dimension table; in the case of the normalized representation, it only adds the roll up functions to the set of relations.

From the above paragraph we conclude that, in general, the execution of the *Generalize, Specialize* and *DelLevel* operators are more expensive in the denormalized representation. On the other hand, the execution of the *Relate* operator is more expensive in the normalized representation, because of the cost incurred in operations of type (c). Because the *Unrelate* operator only performs operations of type (a) and (b), it has the same cost for both representations.

The computation of the *Relate* operator, depicted in Figure 8, is significantly different in both representations. In the denormalized representation, the effect of the operator is basically the addition, of a functional dependency between l_a and l_b to the schema, followed by the computation of the transitive reduction of the dag with the new edge. The two constraints tested are whether l_a is parallel to l_b, and the new functional dependency between $l_a \rightarrow l_b$ added is satisfied by the relation. The transitive reduction is computed in the last for-loop by detecting and deleting all the direct edges which are redundant because of the addition of the edge (l_a, l_b). In the normalized representation, the verification of the functional dependency $l_a \rightarrow l_b$ is done joining some relations in $\mathcal{R}_d$. In order to do that, it is enough to make the joins of the relation in the path from a level in the *Meet* set of l_a and l_b. $Meet(l_a, l_b, S_d)$ computes a set of nodes which reach both l_a and l_b, but which are not pairwise reachable. If the dimension is a lattice the *Meet* of every pair of vertices is unique. The path from any vertex in $Meet(l_a, l_b, S_d)$ to l_a is unique, the same holds for their paths to l_b. It is also easy to see that if the functional dependency is satisfied for the relation resulting from joining those two paths, it is also satisfied by the join of all the relations in $\mathcal{R}_d$. Finally, in the case of a normalized representation, upon deleting edges (when computing the transitive reduction), we must delete from $\mathcal{R}_d$ their associated relations.

Method for Denormalized Rep.	**Method for Normalized Rep.**

precondition: $Parallel(l_a, l_b, S_d)$
precondition: R_d satisfies the functional dependency $l_a \rightarrow l_b$
For every level l_i such that there is an edge from l_b to l_i in S_d and $l_i \in ReachSet(l_a, S_d)$:
  delete edge (l_b, l_i) from S_d
For every l_j such that there is an edge from l_j to l_b in S_d and $l_j \in PreSet(l_a, S_d)$:
  delete edge (l_j, l_b) from S_d
Add edge (l_a, l_b) to S_d
	precondition: $Parallel(l_a, l_b, S_d)$
	Let l_1 is some level in $Meet(l_a, l_b, S_d)$
	$\tau_1 := Path(l_1, l_b, S_d)$
	$\tau_2 := Path(l_1, l_a, S_d)$
	Let R_1 be the join of the relations in τ_1
	Let R_2 be the join of the relations in τ_2
	$R_3 := R_1 \bowtie R_2$
	precondition: R_3 satisfies the functional dependency $l_a \rightarrow l_b$
	For every level l_i such that there is an edge from l_b to l_i in S_d and $l_i \in ReachSet(l_a, S_d)$:
	  delete edge (l_a, l_i) from S_d
	  $\mathcal{R}_d := \mathcal{R}_d \setminus \{R_{l_a l_i}\}$
	For every l_j such that there is an edge from l_j to l_b in S_d and $l_j \in PreSet(l_a, S_d)$:
	  delete edge (l_j, l_b) from S_d
	  $\mathcal{R}_d := \mathcal{R}_d \setminus \{R_{l_j l_b}\}$
	Add edge (l_a, l_b) to S_d
	$\mathcal{R}_d := \mathcal{R}_d \cup \{R_3\}$

Figure 8: Algorithms for $Relate(R_d, l_a, l_b)$.

5.3 Instance Update Operators

The algorithm for the instance update operators has two type of operations: (a) operations that verify the preconditions, and (b) operations that compute the net effect of the updates, i.e., the set of tuples being added and deleted to/from the relation(s). The comparison between execution costs of the algorithm in both representations seems to require a more detailed analysis than in the case of the structural update operators. Let us present some general observations on the implementation proposed here. In the case of the *AddInstance* operator, operations of type (a) mainly consist on verifying that the inserted element reaches the same elements in the levels above in the schema, through any possible path; for the *DelInstance* operator, operations of type (a) mainly consist on verifying that the element deleted is a leaf. In general, these operations should have a higher execution cost in the normalized representation than in the denormalized representation, because the algorithm must perform joins over the normalized relations versus selections over the denormalized relation, respectively. Without considering particular instances, the comparison of the costs of the operations of type (b) for both representations does not give a definitive result favoring any of the alternatives.

The algorithm for the *AddInstance* operator (see Figure 9) first verifies if the new element x is not in the instance set of l; this is performed by the procedure $InInstSet(l, e)$. $InInstSet(l, e)$ computes the instance set of l and verifies if e is in this set. The second part of both algorithms consists in verifying that the elements in p reach the same elements in the common levels above in the DAG. In order to do that, we compute the tuple associated to each element e in p(see operator definition) using the procedure $ReachTuple(l_i, e)$. $ReachTuple(l_i, e)$ computes a tuple with values in the levels reached by l selected from the relation, and nulls in the levels not reached by l. In the denormalized representation, this tuple is computed through a selection and projection over R_d. In the normalized case, performing a traversal of the subgraph of S_d defined by the *Reach Graph*(l, S_d) operator, selecting the elements indirectly reached by $l_i = e$ from the relations in $\mathcal{R}_d$. In each iteration this tuple is merged with the tuple resulting from the merge of the tuples of other elements in p. This merging procedure gives true if the values in common attributes are the same, or one is null. For instance, merging two tuples $(a_1, b_1, null)$ and $(a_1, null, null)$, returns a tuple $(a_1, b_1, null)$. In the denormalized representation, the merged tuple is added to R_d and the tuples associated to elements in p are deleted from R_d because they are no longer leaves of the dimension. In the normalized case, the new element is added to the relations in $\mathcal{R}_d$ associated to the edges which depart from l. In the last two lines of the algorithm we assume that if l rolls up directly to All, $p(All) = all$, and $R_{lAll} := R_{lAll} \cup \{x\}$, because if $l \preceq All$, there cannot be another level l_j s.t. $l_i \preceq l_j$.

Method for Denormalized Representation	**Method for Normalized Representation**

precondition: not $InInstSet(l, x)$
Let t be a tuple s.t. $t(l) = x$ and $t(l_i) = null$ for all other levels.
For every attribute l_i in the tuple p:
  $t_i := ReachTuple(l_i, p(l_i))$
  $R_1 := R_1 \cup \{t_i\}$
  precondition: is it possible to merge t_i with t
  Let t be the resulting tuple of
  merging t with t_i
$R_d := R_d \cup t \setminus R_1$
	precondition: not $InInstSet(l, x)$
	Let t be the *null* tuple.
	For every attribute l_i in the tuple p:
	  $t_i := ReachTuple(l_i, p(l_i))$
	  precondition: is it possible to merge t_i with t
	  Let t be a tuple of the form $(x, p(l_i))$
	  $R_{ll_i} := R_{ll_i} \cup t$

Figure 9: Algorithms for *AddInstance*(R_d, l, x, p).

5.4 Complex Operators

The algorithm for the *Reclassify* operator is presented in Figure 10. Let us explain first the method for the denormalized case. First, it is verified that l_a and l_b are directly connected. Then, we check that the element x_b to which x_a will be reclassified, belongs to the instance set of l_b. This is done using the procedure $InInstSet$. After this, the main task consists on verifying that the dimension after the reclassification will satisfy the functional dependencies of the schema. Let us explain how that precondition is verified: first we compute the tuples t_1 and t_2 using the procedure *ReachTuple*. If there is an alternative path, i.e., a path which do not pass through $l_a l_b$, from a level l below l_a to some level l' reached by l_b, and l has some elements that reach x_a, then the tuples t_1 and t_2 must have the same value in l'. The procedure *FindConfLevels* finds the conflicting levels and stores them in a variable $ConfLevels$. The main algorithm verifies then, that the tuples t_1 and t_2 have the same value in the attributes in $ConfLevels$. After that, the update is performed.

In the normalized approach, the computation of *ReachTuple* could be expensive because it requires a traversal of the subgraph of S_d induced by the vertices in $Reach(l_a, S_d)$, making a sequence of selections over the relations. It is not necessary to compute all the the values in the attributes of t_1 and t_2, but only the values in attributes in $ConfLevels$. Further, we use the fact that we only need to compare the attributes in the conflict levels which are minimal, i.e., the levels in $ConfLevels$ which are not reachable by other levels in $ConfLevels$. We store in LTC the minimal levels of $ConfLevels$. Then, the path from l_a to every every level l_i in LTC is computed. This path is required by the procedure *ReachTuple*. *ReachTuple* performs the traversal of the path making a sequence of selection over the relations in order to compute the value associated to l_i. Note that the *ReachTuple* procedure requires the path τ, along which the computation must proceed, to be specified as one of its arguments. However, we prefer to overload the method, rather than changing its name.

The computation of $ConfLevels$ is performed traversing the graph below l_a by the recursive procedure $FindConfLevels$. The vertices which are candidate to be added to $ConfLevels$ are stored in A_1, and are determined by the operation $Joint(l_i, l_b, S_1)$ which computes the vertices reached by both l_i and l_b in S_1 which are not pairwise reachable. The procedure $FindConfLevels$ performs the minimum amount of outer-joins to build the relation R; the relation R is required to determine whether there are null values in the level which reach the levels in A_1. If there are not null values, the levels in A_1 are added to $ConfLevels$. The set of levels to visit are pruned (sentence $Visited := Visited \cup PreSet(l_i, G_1)$) using the fact that if there are no elements which reach x_a from a level l then there are no elements which reach x_a from levels below l.

The computation of the update over the relation when the precondition is computed is straightforward.

Method for Denormalized Representation	**Method for Normalized Representation**

Main
precondition: there is an edge (l_a, l_b) in S
precondition: $InInstSet(l_b, x_b)$
$t_1 := ReachTuple(l_a, x_a)$
$t_2 := ReachTuple(l_b, x_b)$
Call *LevelsToCheck*
For every level $l \in LTC$:
  precondition: $t_1(l) = t_2(l)$
$R_d := \mu(R_d, l_a = x_a, t_2)$
 	
Procedure *FindConfLevels*
$R_1 := PreRel(l_a, x_a)$
Let S' be the dag resulting from deleting the edge (l_a, l_b) from S_d
For every level l_i in $PreSet(l_a, S_d)$:
  $A_1 := Join(l_i, l_b, S')$
  If $(Count(\sigma_{l_i \neq null} R_1) > 0)$
  then $ConfLevels := ConfLevels \cup A_1$
	Main
	precondition: $Reach(l_a, l_b, S)$
	precondition: $InInstSet(l_b, x_b)$
	Let S_1 be the dag resulting from deleting the edge (l_a, l_b) from S_d
	Let G_1 be the subgraph of S induced by the vertices in $PreSet(l_a, S_d)$
	$R := \sigma_{l_a = x_a} R_{l_a l_b}$
	Call *FindConfLevels*(l_a)
	$LTC := MinSet(ConfLevels, S_d)$
	For every level $l_i \in LTC$:
	  $\tau_i := Path(l_b, l_i, S_d)$
	  $t_i := ReachTuple(\mathcal{R}_d, \tau_i, l_b = x_b)$
	  $t_j := ReachTuple(\mathcal{R}_d, \tau_i, l_b = R_{l_a l_b}(x_a))$
	  precondition: $t_i(l_i) = t_j(l_i)$
	$R_{l_a l_b} := \mu(R_{l_a l_b}, l_a = x_a, (l_b, x_b))$
	
	Procedure *FindConfLevels*(l_1)
	$Visited := Visited \cup \{l_1\}$
	For every level l_i in $PreSet(l_1, G_1) \setminus Visited$
	  $A_1 := Joint(l_i, l_b, S_1)$
	  If $A_1 \nsubseteq ConfLevels$
	  $R := R\ outerjoin\ R_{l_i l_1}$
	  If $Count(\pi_{l_i} \sigma_{l_1 \neq null} R) = 0$:
	    $Visited := Visited \cup PreSet(l_i, G_1)$
	    Else $ConfLevels := ConfLevels \cup A_1$
	Call *FindConfLevels*(l_i)

Figure 10: Algorithm for $Reclassify(R_d, l_a, l_b, x_a, x_b)$.

Example 9 *Let us apply the algorithm of figure 10 to the dimension depicted in figure 5.*
For the denormalized *case, the corresponding relation representing the dimension instance would be:*

I	K	A	B	C	E	F	G
i_1	k_1	a_1	b_1	c_1	e_1	f_1	$g1$
i_2	k_1	a_2	b_2	c_2	e_2	f_2	$g1$
i_3	k_1	a_2	b_2	c_2	e_2	f_2	$g1$
i_4	k_2	a_3	b_2	c_2	e_2	f_2	$g1$
i_5	k_2	a_4	b_2	c_2	e_2	f_2	$g1$

Let us suppose we would like to reclassify a_2 from b_2 to b_1. After the first validations, we compute $Reachtuple(A, a_2)$

and $Reachtuple(B, b_1)$ *, obtaining:*
$t_1 =< null, null, a_2, b_2, c_2, e_2, f_2, g1 >$;
$t_2 =< null, null, null, b_1, c_1, e_1, f_1, g1 >$.
Procedure FindConfLevels *computes* $R_1 = PreRel(A, a_2)$, *yielding*

I	A
i_2	a_2
i_3	a_2

The conflicting level set will be the result of $Join(I, B, S')$, *where* S' *is the graph* S *without the edge joining levels* A *and* B, *in order to compute the alternative paths not including* l_a *and* l_b. *Then,* $LTC = \{C\}$. *As* $t_1(C) \neq t_2(C)$, *the proposed reclassification fails.*

Due to the lack of space, we will not give the complete example for the normalized case, but we will comment on it. S_1 *will be the same graph as* S' *above, obtained deleting the edge* A, B *from* S. G_1 *will be the graph s.t.* $V = \{A, I\}$, *and* $E = \{I - A\}$, *this is, the graph s.t. its edges are all the edges reaching* A, *in this example, only* $I - A$. *Applying the procedure* $LevelsToCheck(A)$ *will only require one recursive call, with argument* I. *Eventually, we get* $LTC = \{C\}$. *The* $ReachTuple$ *procedure will have* $\tau =< B, C >$ *as one of its arguments. Thus, two tuples will be computed:*
$t_1 = ReachTuple(\mathcal{R}_d, < B, C >, B = b_1)$
$t_2 = ReachTuple(\mathcal{R}_d, < B, C >, B = R_{AB}(a_2))$

The tuple $t_1 =< b_1, c_1 >$ *is obtained directly from* R_{BC}, *while* $t_2 =< a_2, b_1, c_2 >$ *is obtained joning* R_{AB} *and* R_{BC}, *as indicated in the path used as argument of* ReachTuple. *Then, as* $t_1(C) \neq t_2(C)$, *the reclassification cannot proceed.*

6 Conclusion and future work

Data warehouse evolution has foucused on fact tables(when we represent it as a set of dimensions and fact tables). However, we showed that dimensions, being an important portion of the information available in a data warehouse, are also subjected to evolve over time. In a former work [6] presented a multidimensional model which addresses the problem of dimension updates in data warehouses, as well as a set of operators for performing such updates. In this paper we have extended the model with a set of semantically meaningful operators as *reclassification, merge, split,* and *update*. Although these operators could be implemented by means of a transaction involving only basic operators, they are more efficiently performed as a single one. We also formally defined the conditions under which they lead to a unique consistent dimension.

We also propose a mapping from the dimensional to the relational model using two approaches: denormalized and normalized relational tables (Star or Snowflake schemas, respectively [8]). Some interesting properties verified by this mapping are presented. Finally, we compare of both approaches w.r.t. the dimension update operators. It has been an accepted fact that denormalized dimension tables perform better than normalized ones. However, no formal or empiric evidence on this assertion had been provided so far.

Although several works had been performed on modeling for OLAP applications [1, 5, 4], only [3] and [10] deal with dimension modeling, in a rather formal fashion. However, they do not consider dimension updates, and we think our approach formalizes dimension modeling more accurately.

We must point out certain limitations of the operators we propose, over which we are currently working on: (a) the operators only deal with one update at a time. We would like to be able to perform a bulk reclassification, for instance. Moreover, we would also like to be able to express these operations over elements satisfying some condition; (b) we express the rollup functions extensionally. It would be an advantage, from the user's point of view, to do this intensionally . Finally, we are working on taking advantage of the multidimensional model and the dimension update operators, in order to develop a model which takes into account the evolution of the data warehouse over time.

References

[1] R Agrawal, A Gupta, and S. Sarawagi. Modeling multidimensional databases. *IBM Almaden Research Report*, 1997.

[2] A. V. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM J. Computing 1:2, pgs. 131-137*, 1972.

[3] L. Cabibbo and R. Torlone. A logical approach to multidimensional databases. In *EDBT'98: 6th International Conference on Extending Database Technology*, pages 253–269, Valencia, Spain, 1998.

[4] A. Golfarelli, D. Maio, and S. Rizzi. Conceptual design of data warehouses from e/r schemes. In *Proceedings of the Hawai International Conference on System Sciences*, Kona, Hawai, 1998.

[5] M. Gyssens and L. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of the 22nd VLDB Conference*, pages 106–115, Bombay, India, 1996.

[6] C. Hurtado, A.O. Mendelzon, and A. Vaisman. Maintaining data cubes under dimension updates. *Proceedings of IEEE/ICDE99*, 1999.

[7] Informix Corporation. *Informix OnLine Extended Parallel Server and Informix Universal Server : A New Generation of Decision-Support Indexing for Enterprisewide Data Warehouses*, 1996. White Paper.

[8] R. Kimball. *The Data Warehouse Toolkit.* J.Wiley and Sons, Inc, 1996.

[9] W. Lehner. Modeling large olap scenarios. In *Proceedings of the 1998 International Conference on Extending Database Technology*, Valencia, Spain, 1998.

[10] W. Lehner, J. Albretch, and H. Wedkind. Normal forms for multidimensional databases. In *Technical Report*, University of Erlangen-Nuremberg, Germany, 1998.

[11] D. Lomet and Editors Widom, J. *IEEE Data Engineering Bulletin. Special Issue on Materialized Views and Data Warehousing*, June 1995.

[12] Pilot Software. *An introduction to OLAP*, 1996. White Paper.

[13] Corey Salka. Ending the rolap/molap debate: Usage based aggregation and flexible holap. In *Proceedings of the IEEE 14th ICDE*, Orlando,Florida, 1998.

[14] A. Shoshani. Olap and statistical databases: Similarities and differences. In *Proceedings of ACM TODS*, 1997.

[15] J Widom. Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, 1995.

A Implementation of the Update Operators

A.1 Structural Update Operators

Generalize The algorithms for the *generalize* operator are presented in Figure 11. First, we verify that the new level l_n is not in the schema of the relational representation. Then, we check that the new roll up function, represented using the relation F, connects every element in l with some element in l_n. This is indicated in the second line of the denormalized algorithm. In this algorithm, we join the new roll up table with the pre-existent dimension table. The normalized algorithm must consider the case in which the level l has no levels above in the relational schema, i.e., it is connected to *All* in the dimension schema. This is one of the reasons of adding the relation $R_{l_i All}$ to the mapping. In this fashion, we can treat every case uniformly.

Method for Denormalized Representation	Method for Normalized Representation
$not\ In(l_n, S_d)$? $\pi_l R_d \subseteq \pi_l F$? $S_d := AddEdge(S_d, (l, l_n))$ $R_d := R_d \bowtie F$	$notIn(l_n, S_d)$? Let l_i be some level in $ReachSet(l, S_d)$ $\pi_l R_{ll_i} \subseteq \pi_l F$? $S_d := AddEdge(S_d, (l, l_n))$ $\mathcal{R}_d := \mathcal{R}_d \cup \{F\}$

Figure 11: Methods for $Generalize(R_d, l, l_n, F)$.

Specialize The algorithms for the *specialize* operator, presented in Figure 12, start verifying that the new bottom level is not in the schema of the relational representation. Then, they make sure that the third constraint of definition 2 holds(i.e., that no tuple of F has no associated element in the former bottom level). In the case of normalized representation, it is necessary to perform a join between F and some relation having the old bottom level as attribute. We finally add $\mathcal{R}_d$ to the result of this join; in the denormalized representaion, F is joined with R_d.

Method for Denormalized Representation	**Method for Normalized Representation**
$not\ In(l_n, S_d)$? $S_d := AddEdge((l_n, l_{inf}), S_d)$ $R_d := F \bowtie R_d$	$notIn(l_n, S_d)$? Let l_i be some level in $ReachSet(l_{inf}, S_d)$ $F_1 := \pi_{l_{inf}, l_n}(F \bowtie R_{l_{inf}, l_i})$ $S_d := AddEdge(S_d, (l_n, l_{inf}))$ $\mathcal{R}_d := \mathcal{R}_d \cup \{F_1\}$

Figure 12: Methods for $Specialize(R_d, l_n, F)$.

Delete Level The algorithm for the *delete level* operator is depicted in Figure 13. First, it is verified that the deletion of the level does not yield a dimension with several bottom levels. After that, the new dimension schema is computed through the addition of the edges which connect vertices directly below l with vertices directly above it, if there are no other paths between them. In the denormalized case, the resulting dimension is computed by means of a projection over the remaining attributes. For the normalized case, the relations associated to the edges being added are computed and added to $\mathcal{R}_d$ by means of a join. Notice that if the deleted level is one directly connected to All, we have to project the result of the join, only into the levels in $Preset(l, S_d)$. Finally, the relations at the edges associated to l are dropped from $\mathcal{R}_d$.

Method for Denormalized Representation	**Method for Normalized Representation**

$PreSet(l, S_d) \neq \phi$ or $\lvert ReachSet(l, S_f)\rvert = 1$?
$S_d := DelVertex(l, S_d)$
For every l_i in $PreSet(l, S_d)$
  For every l_j in $ReachSet(l, S_d)$
    If not $l_i \in ReachSet^*(l_j, S_d)$
      $S_d := AddEdge((l_i, l_j), S_d)$
$R_d := \pi_{A_d \setminus \{l\}} R_d$
	$PreSet(l, S_d) \neq \phi$ or $\lvert ReachSet(l, S_f)\rvert = 1$?
	$S_d := DelVertex(l, S_d)$
	For every l_i in $PreSet(l, S_d)$
	  For every l_j in $ReachSet(l, S_d)$
	    If not $l_i \in ReachSet^*(l_j, S_d)$
	      $S_d := AddEdge((l_i, l_j), S_d)$
	If $l_j = All$
	      $R_{l_i l_j} := \pi_{l_i}(R_{l_i l} \bowtie R_{l l_j})$
	Else
	      $R_{l_i l_j} := \pi_{l_i l_j}(R_{l_i l} \bowtie R_{l l_j})$
	      $\mathcal{R}_d := \mathcal{R}_d \cup \{R_{l_i l_j}\}$
	For every l_i in $PreSet(l, S_d)$
	  $\mathcal{R}_d := \mathcal{R}_d \setminus \{R_{l_i l}\}$
	For every l_j in $ReachSet(l, S_d)$
	  $\mathcal{R}_d := \mathcal{R}_d \setminus \{R_{l l_j}\}$

Figure 13: Algorithms for the operator $DelLevel(R_d, l)$.

Unrelate As we showed in Section 2, the *unrelate* operator makes l_a parallel to l_b, maintaining the indirect relation between the edges above l_a, and below l_b. The algorithms depicted in Figure 14, basically compute the new edges between the levels directly above l_b and directly below l_b, which must be added. In the normalized case, the algorithm also adds to $\mathcal{R}_d$ the relations associated to those edges.

Method for Denormalized Rep.	Method for Normalized Rep.

$S_d := DelEdge((l_a, l_b), S_d)$
For every l_i in $PreSet(l_a, S_d)$:
  If l_b not in $Reach^*(l_i, S_d)$
    $S_d := AddEdge((l_i, l_b), S_d)$
For every l_i in $Reach(l_b, S_d)$:
  If l_i not in $Reach^*(l_a, S_d)$
    $S_d := AddEdge((l_b, l_i), S_d)$
	$S_d := DelEdge((l_a, l_b), S_d)$
	For every l_i in $PreSet(l_a, S_d)$:
	  If l_b not in $Reach^*(l_i, S_d)$
	    $S_d := AddEdge((l_i, l_b), S_d)$
	    $\mathcal{R}_d := \mathcal{R}_d \cup \{R_{l_i l_b}\}$
	For every l_i in $Reach(l_b, S_d)$:
	  If l_i not in $Reach^*(l_a, S_d)$
	    $S_d := AddEdge((l_a, l_i), S_d)$
	    $\mathcal{R}_d := \mathcal{R}_d \cup \{R_{l_a l_i}\}$
	$\mathcal{R}_d := \mathcal{R}_d \setminus \{R_{l_a l_b}\}$
	
	Note:
	$R_{l_i l_b} := \pi_{l_i, l_b}(R_{l_i l_a} \bowtie R_{l_a l_b})$
	if $l_i \neq All$
	$R_{l_a l_i} := \pi_{l_a, l_i}(R_{l_a l_b} \bowtie R_{l_b l_i})$
	otherwise
	$R_{l_a l_i} := \pi_{l_a}(R_{l_a l_b} \bowtie R_{l_b l_i})$

Figure 14: Algorithms for the operator $Unrelate(R_d, l_a, l_b)$.

A.2 Instance Update Operators

DelInstance The algorithm for the delete instance operator is presented in Figure 15. Both methods verify that the element deleted is a leaf of the dimension instance. This is performed by the procedure *IsLeaf* depicted in Figure 19, which differs in both representations. In addition, when deleting an element, the algorithm for the denormalized representation deletes the tuples associated to the deleted element, and adds the tuples associated to the new leaves. The algorithm for the normalized representation only deletes the tuples with $l = x$ in the relations associated to edges which depart from l.

Method for Denormalized Representation	Method for Normalized Representation

$IsLeaf(l, x)$?
$R_1 := \sigma_{l=x} R_d$
Let t_i be the *null* tuple
For every l_i in $ReachSet(l, S_d)$:
  If $IsLeaf(l_i, R_1(l_i)$ then
    $t_i := ReachTuple(l_i, R_1(l_i))$
    $R_2 := R_2 \cup \{t_i\}$
$R_d := R_d \setminus R_1 \cup R_2$
	$IsLeaf(l, x)$?
	For every l_i in $ReachSet(l)$:
	  $R_{ll_i} := \pi_{l \neq x} R_{ll_i}$

Figure 15: Algorithms for $DelInstance(R_d, l, x)$.

Update The methods for the *update* operator first verify that the new element is not in the instance set of l using the procedure *InInstSet*(Figure 19). Then, in the denormalized representation, R_d is updated. Analogously,all the relations with x as attribute are updated in the normalized case.

A.3 Complex Operators

Merge The algorithm for merging a set of elements in a level l into a single one, considers the case in which all the elements to be merged roll up to the same element in all the levels l_i such that $l \preceq l_i$. In the two approaches, the algorithm first verifies that the new element does not belong to the instance set of the level. This is performed by the method $InInstSet(l, e)$. Then, as we stated in section **??**, as we consider the case in which all the merged

Method for Denormalized Rep.	Method for Normalized Rep.
not $InInstSet(l, x_n)$? $Count(\sigma_{l=x_n} R_d) = 0$? $R_d := \mu(R_d, (l = x), (l, x_n))$	not $InInstSet(l, x_n)$? For every l_i in $PreSet(l, S_d)$: $R_{l_i l} := \mu(R_{l_i l}, l = x, (l, x_n))$ For every l_j in $ReachSet(l, S_d)$: $R_{ll_j} := \mu(R_{ll_j}, l = x, (l, x_n))$

Figure 16: Algorithms for $Update(R_d, l, x, x_n)$.

elements roll up to the same element in the levels in $Reachset(l, S_d)$, the algorithm checks this condition. Note here that condition $L_1 \neq \phi$ does not include the distinguished level *All*. Then the algorithms proceed in the following fashion: in the denormalized approach, the algorithm simply checks the condition stated at the beginning of this paragraph, performing a selection over the relation, with the condition $l \in E$, which selects the tuples s.t. their attribute l has a value which is in the set of elements to be merged (E). These tuples are then updated, setting the attribute l with the new value e. In the normalized approach, the algorithm is analogous. The main difference resides in the fact that the verification is performed over as many relations as levels reached from l. The update must be done over these relations, and also over the relations including attributes representing levels that reach l.

Example 10 *Let us suppose the dimension depicted in figure 1, where we want to merge sub-regions sr_2 and sr_3 into a sub-region we will denote sr_5. The relational representation in the denormalized approach is :*

S	Sr	R	C
s_1	sr_1	r_1	$c1$
s_2	sr_2	r_2	$c2$
s_3	sr_3	r_2	$c2$
s_4	sr_3	r_2	$c3$
s_5	sr_4	r_3	$c3$

According to algorithm of figure 10, we first check that sr_5 is not in the table above, in column Sr.ReachSet(Sr,S_d) *returns the set $\{R\}$. The step $Count(\pi_R \sigma_{Sr \in \{sr_2, sr_3\}} R_d) = 1$ gives* true. *Then, we must update column R, and the table will look like the following:*

S	Sr	R	C
s_1	sr_1	r_1	$c1$
s_2	sr_5	r_2	$c2$
s_3	sr_5	r_2	$c2$
s_4	sr_5	r_2	$c3$
s_5	sr_4	r_3	$c3$

As the normalized approach is quite similar, we will only show a pair of tables:

S	Sr
s_1	sr_1
s_2	sr_2
s_3	sr_3
s_4	sr_3
s_5	sr_4

Sr	R
sr_1	r_1
sr_2	r_2
sr_3	r_2
sr_4	r_3

We perform the same operations, on the first table, but we must update both tables (note that *PreSet(Sr,S_d)*={S}). Thus, the tables will become :

S	Sr
s_1	sr_1
s_2	sr_5
s_3	sr_5
s_4	sr_5
s_5	sr_4

Sr	R
sr_1	r_1
sr_5	r_2
sr_4	r_3

Method for Denormalized Representation	**Method for Normalized Representation**
not $InInstSet(l,e)$? $L_1 := ReachSet(l, S_d)$ If $L_1 \neq \phi$: $Count(\pi_{L_1}\sigma_{l\in E}R_d) = 1$? $R_d := \mu(R_d, (l \in E), (l,e))$	not $InInstSet(l,e)$? $L_1 := ReachSet(l, S_d)$ If $L_1 \neq \phi$: For every l_i in L_1 $Count(\pi_{l_i}\sigma_{l\in E}R_{ll_i}) = 1$? For every l_i in $PreSet(l, S_d)$: $R_{l_i l} := \mu(R_{l_i l}, l \in E, (l,e))$ For every l_j in $ReachSet(l, S_d)$: $R_{ll_j} := \mu(R_{ll_j}, l \in E, (l,e))$

Figure 17: Algorithm for $Merge(R_d, l, E, e)$.

Split The algorithm for splitting an element into a set of elements specified by the user, begins, as the former one, by checking that the new elements do not belong to the instance set of the level, in both approaches. The next step consists in checking that the proposed split is possible, this is, leads to a consistent dimension. The main difference resides on how to compute the initial relation R_1 (see figure A.3). In the normalized case, the elements reaching the element to be split (e), are computed by a single projection. In the normalized case, we must compute these elements across a series of tables, with algorithm $PreRel(l,e)$. We use a chase-like procedure for this. Finally, in the denormalized case, we perform a modified outer join(in this variant of the outer join, R(AB) outer S(BC) has schema ABC) between R_1 and each relation representing a split function($F_{l_i l}$). There is a split function for every level reaching l through a single edge. If it turns out to be impossible to merge the tuples in the outer join, the split will not be admitted. This happens when a *null* value cannot be eliminated from the column l' in R_2, meaning that there will be an inconsistency in the new instance(see example below). If the merging is possible, we delete the tuples with $l = e$, and add the result of the outer join. In the normalized approach, we proceed in an analogous fashion. The main difference is that we must build the resulting tables for each relation including levels above l, deleting the tuples which include e, and adding the cartesian product depicted in figure A.3. For the tables including attributes for levels below l, a join is performed between each split function and the corresponding relation.

Example 11 *Let us return to figure 6, but assume that we propose that c_2 will be assigned to d_2, leading to an inconsistent dimension. In the denormalized approach, we will have the following instance for R_1 :*

A	B	C	E	F
a_1	b_1	c_1	e_1	f_1
a_2	b_2	c_2	e_1	f_1
a_3	b_3	c_3	e_1	f_1

The split functions will be F_{BD} and F_{CD}:

B	D
b_1	d_1
b_2	d_1
b_3	d_2

C	D
c_1	d_1
c_2	d_2
c_3	d_2

The two outer joins, $(R_1 \; outerjoin \; F_{BD}) \; outerjoin \; F_{CD})$, produce the table :

A	B	C	D	E	F
a_1	b_1	c_1	d_1	e_1	f_1
a_2	b_2	*null*	*null*	e_1	f_1
a_3	b_3	c_3	d_2	e_1	f_1
null	*null*	c_2	d_2	*null*	*null*

We see that it will be impossible to merge the tuples. This is a consequence of the fact that, in the proposed splitting, a_2 reaches d_1 and d_2, according to the chosen path. If the splitting were valid, we should delete the three tuples in R_d, and add the three tuples resulting from the outer join.

Method for Denormalized Rep.	**Method for Normalized Rep.**
not $InInstSet(E, l)$? $R_1 := \pi_{A_d \setminus \{l\}} \sigma_{l=e} R_d$ $R_2 := R_1$ For every function $F_{l_i l}$ in P: Rename l as l'_i in F $R_2 := R_2 \; outerjoin \; F_{l'_i l}$ For every tuple t_i in R_2: is it possible to merge the values of every l'_i? Merge the values in attr. l $R_d := R_d \setminus \sigma_{l=e} R_d \cup \pi_{A_d} R_2$	not $InInstSet(E, l)$? $R_1 := PreRel(l, e)$ For every function $F_{l_i l}$ in P: Rename l as l'_i in F $R_1 := R_1 \; outerjoin \; F_{l'_i l}$ For every tuple t_i in R_1: is it possible to merge the values of every l'_i? For every l_i in $ReachSet(l, S_d)$: $R_4 := E \times \pi_{l_i} \sigma_{l=e} R_{ll_i}$ $R_{ll_i} := R_{ll_i} \setminus \sigma_{l=e} R_{ll_i} \cup R_4$ For every l_i in $PreSet(l, S_d)$: $R_4 := \pi_{R_{ll_i}.l_i, F_{l_i l}.l}(R_{ll_i} \bowtie F_{l_i l})$ $R_{l_i l} := R_{l_i l} \setminus \sigma_{l=e} R_{ll_i} \cup R_4$

Figure 18: Algorithm $Split(R_d, l, e, E, P)$.

Example 12 *In example 6, algorithm $PreRel(l, e)$, for the normalized case, would proceed as follows. The initial tableaux would be :*

A	B	C
a_1	b_1	*null*
a_2	b_2	*null*
a_3	b_3	*null*
a_1	*null*	c_1
a_2	*null*	c_2
a_3	*null*	c_3

Applying the functional dependencies $A \to B, A \to C$, we obtain the desired relation, containing the three tuples.

Procedure	**Method for Denormalized Representation**	**Method for Normalized Representation**
$InInstSet(l,e)$	$Count(\sigma_{l=e}R_d) = 0$?	Let l_1 be some level in $ReachSet(l,S_d)$ $Count(\sigma_{l=e}R_{ll_1}) = 0$?
$IsLeaf(l,e)$	$R_1 := \sigma_{l=e}R_d$ For every l_i in $PreSet(l,S_d)$: $R_1(l_i) = null$?	For every level l_i in $Preset(l)$: $Count(\sigma_{l=x}R_{l_il}) = 0$?
$ReachTuple(l,e)$	$t := \pi_{ReachSet*(l,S_d)}\sigma_{l=e}R_d)$ return(t)	Let $G := ReachGraph(l,S_d)$ Traverse G selecting the values of t from the relations of the edges starting with $l=e$ return(t)
$PreRel(l,e)$	$R := \pi_{PreSet*(l,S_d)}\sigma_{l=e}R_d)$ return(R)	Build a tableaux T with all the attributes representing nodes in the graph $G = PreGraph(l,S_d)$, where $PreGraph(l,S_d)$ is the subgraph of S_d containing all the nodes in $Preset^*(l) \setminus \{l\}$. The initial values in T will be the tuples, for each R_i, in $\sigma_{l=e}R_i$, completed with nulls. This is, $card(T) = \sigma card(R_i)$, where R_i is s.t. l is an attribute of its schema. As the graph corresponds to a functional dependency, chase T with the set formed by such dependencies. The resulting tableaux is PreRel(l,e).

Figure 19: Common procedures.