Updating OLAP Dimensions

Carlos A. Hurtado
chl@db.toronto.edu
University of Toronto

Alberto O. Mendelzon
mendel@db.toronto.edu
University of Toronto

Alejandro A. Vaisman
av2n@dc.uba.ar
Universidad de Buenos Aires

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 can take place either at the structural level (e.g. addition of categories or modification of the hierarchical structure) or at the instance level (elements can be inserted, deleted, merged, etc.). They are poorly supported (or not supported at all) in current commercial systems and have not been addressed in the literature. In a previous paper we introduced a formal model supporting dimension updates. Here, we extend the model, adding a set of semantically meaningful operators which encapsulate common sequences of primitive dimension updates in a more efficient way. We also formally define two mappings (normalized and denormalized) from the multidimensional to the relational model, and compare an implementation of dimension updates using these two approaches.

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).

DOLAP 99 Kansas City Mo USA
Copyright ACM 1999 1-58113-220-4/99/11...$5.00

Several works have proposed formal models for OLAP applications [1, 3, 2]. Most of these are based on the original idea of the "star schema," [7] 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 [10], 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 [6, 9]), a redefinition of the initial requirements may cause a dimension update.

In a previous work [5] we introduced a multidimensional model that includes a framework for dimension updates and a set of primitive dimension update operators. Here we present a set of semantically meaningful complex operators that encapsulate common sequences of primitive ones, leading to a more efficient implementation. We also sketch algorithms for implementing both the primitive and the complex operators in a ROLAP star schema environment, and compare two different approaches, normalized and denormalized dimension tables, from the point of view of dimension update algorithms.

The contributions of this paper are: (a) an extension of the multidimensional model formerly introduced [5], with a set of *complex operators* which can be implemented more efficiently than equivalent sequences of primitive operators; (b) a mapping from the multidimensional to the relational model, using two different approaches: normalized and denormalized dimension tables, and some properties of these mappings;(c) an algorithmic definition of the dimension update operators, which we use to compare how the two relational representations behave under dimension updates.

The rest of the paper is organized as follows : In Section 2 we review the multidimensional model supporting dimension updates, and we introduce the new update operators. In section 3 we present and discuss the implementation of the dimension update operators using two possible relational representations for the multidimensional model: normalized and denormalized dimension tables. We conclude in Section 4, summarizing the paper and commenting on our future research directions.

2 Dimension Update Operators

2.1 Dimension Modeling

In this section, we give an overview of the data model which was introduced in a former work [5]. We refer the interested reader to that work, where she/he will find a complete description of the model.

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 there is no level* l_c *distinct from* l_a *and* l_b*, s.t.* $l_a \preceq^* l_c$ *and* $l_c \preceq^* l_b$*. 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 *rollup functions*, one for each edge in the dimension schema. An important feature of the model is that it allows the representation of so-called "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 rollup function* $RUP_{l_1}^{l_2} : dom(l_1) \rightarrow dom(l_2)$ *in* RUP*; (2) For each triple of levels* $l_1, l_2, l_3 \in L$ *such that* $l_1 \preceq l_2$ *and* $l_2 \preceq l_3$*,* $ran(RUP_{l_1}^{l_2}) \subseteq dom(RUP_{l_2}^{l_3})$*; and (3)* $DESC$ *is a partial function with signature* $DESC_l^a : dom(l) \rightarrow dom(a)$ *, for each level* l *and attribute* a*, such that* $a \gg l$*.*

Definition 3 (Consistent Dimension Instance) *We say that a dimension instance* $(D, RUP, DESC)$*, defined as above, is* consistent*, if for each pair of paths in the graph with nodes* l_i *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* $RUP_{l_1}^{l_2} \circ \ldots \circ RUP_{l_{n-1}}^{l_n} = RUP_{l_1}^{l'_2} \circ \ldots \circ RUP_{l'_{m-1}}^{l_m}$.

In our model, we can summarize facts through levels, but not necessarily through attributes. The summarizability of facts through levels is always possible by the following constraints presented definitions 2 and 3:

- The range of every rollup function is a subset of the domain of the rollup functions 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 rollup functions must be the same. This ensures that in the common final level, each element belongs to a single class.

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 $< l_m \ldots ln >$ is a path from l_m to l_n in the graph with nodes in L and edges in $\preceq$, $RUP_{l_m}^{l_n} = RUP_{l_m}^{l_{m+1}} \circ \ldots \circ RUP_{l_{n-1}}^{l_n}$.

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.

2.2 Primitive Update Operators

A summary of the primitive set of operators which we introduced in a former work [5] is presented in Figure 1. Operators dealing with attributes are straightforward, and we omit them.

2.3 Complex Update 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*

Operator	Description
Generalize	Adds a new level above a preexisting one, with a rollup function between the old and the new levels.
Specialize	Adds a new level below the current bottom level, which will become the new bottom level (l_{inf}). A rollup function between them is also added.
Relate	Adds a new edge, between two parallel levels. The associated rollup function, if it exists, is determined automatically. If it is not possible to do so uniquely, the operator is not applicable.
Unrelate	Deletes an edge between two levels.
Delete Level	Deletes a level with the precondition that the new hierarchy must have a unique bottom level(*All* cannot be deleted).
Add Instance	Adds a value, call it x, to the domain of some rollup function. A pair of the form (x, y) for each rollup function sharing this domain must be provided as an argument, s.t. y is a value in the the range of each of such rollup functions.
Delete Instance	Deletes a value x from a level l, as well as all the pairs(x, y) s.t. y is an element in the range of a rollup function departing from l.

Figure 1: Primitive Dimension Update Operators.

operators, intended to capture such common sequences and encapsulate them in a single operation. The complex operators are *Reclassify, Split, Merge*, and *Update.* They are all what we call *instance updates*, because they affect the instances of a dimension, not its schema. We will treat each one separately.

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

Operator 1 (Reclassify) *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.*

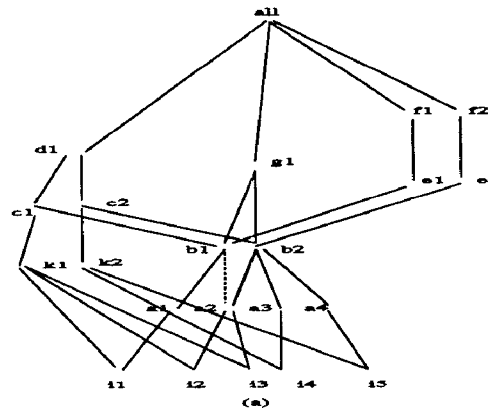

Figure 2: Reclassification

Reclassify is not defined for every possible dimension, as is shown in Figure 2 . Here, if we reclassify a_2 from b_1 to b_2, consistency would be violated at level C, which is reached from l_b and from l_{inf}. However, note that as no level other that B rolls up to level E, no problems arise from this level. This observation leads to a general condition under which reclassification may always be performed.

Definition 4 (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 1. 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* $instset(l_i)$. *A conflicting level is* minimal *if it is not reachable by any other conflicting level.*

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

Split Suppose, for instance, that 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, an operator which lets the user specify which salesperson is assigned to each region, and automatically reorganize the dimension, keeping it consistent. Formally:

Operator 2 (Split) *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* $x_{a_i} \in dom(l_a) \setminus instset(l_a)$, *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* $l_i \preceq l_a, i = 1..m$, $list_i$ *is a list of elements in* $instset(l_i)$ *of the form* $(x_1, \ldots, x_k)$ *s.t.* $RUP^{l_a}_{l_i}(x_i) = x_a$; $\mathrm{Split(d, l_a, x_a, E, P)}$ *is a new dimension* $((dname, L, A, \preceq, \gg), RUP', DESC)$, *where :* $RUP'^{l_a}_{l_i} = RUP^{l_a}_{l_i} \setminus \{(x_i, x_a) | RUP^{l_a}_{l_i}(x_i) = x_a \cup$

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

Example 1 *Suppose we want to split element* d *in Figure 3, into two elements* d_1 *and* d_2 *in the domain of level* D. *The operation will be denoted by* 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 rollup functions corresponding to the new values* d_1 *and* 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.

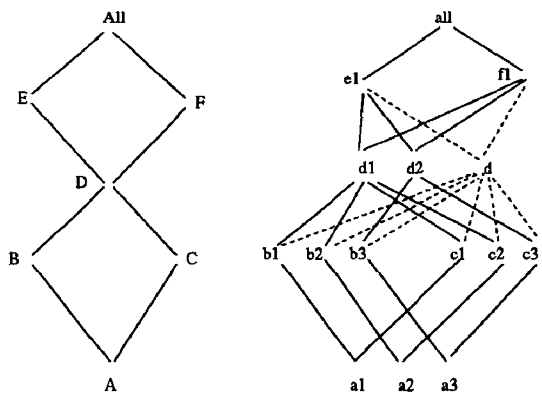

Figure 3: Split operator.

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 3 (Merge) *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* $X = \{x_1 \dots x_n\} \in instset(l_a)$ *s.t. all the elements* $x_i \in X$ *rollup to the same element in every level* l *s.t.* $l_a \preceq l$; Merge(d, l_a, X, x_N) *is a new dimension* $((dname, L, A, \preceq, \gg), RUP', DESC)$, *where :* $RUP'^{l_a}_{l_i} = RUP^{l_a}_{l_i} \setminus \{(x_i, x_j) | RUP^{l_a}_{l_i}(x_i) = x_j, x_j \in X\} \cup \{(x_i, x_N) | RUP^{l_a}_{l_i}(x_i) = x_j, x_j \in X\}$; $RUP'^{l_j}_{l_a} = RUP^{l_j}_{l_i} \setminus \{(x_i, x_j)\} \cup \{(x_N, x_j)\}, x_i \in X, RUP^{l_j}_{l_a}(x_i) = x_j, l_a \preceq l_j$; $RUP'^{l_j}_{l_i} = RUP^{l_j}_{l_i}$, *for all other levels* l_i, l_j. *The new dimension remains consistent.*

Example 2 *Figure 4 shows an example of Merge. We can see that the operation* Merge(d, E, $\{e_2, e_3\}$, X_n) *keeps the dimension in a consistent state.*

Note that *Merge* and *Split* are symmetric, that is, $Merge(Split(d)) = d$. For instance, in Figure 3, if after the split we 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.

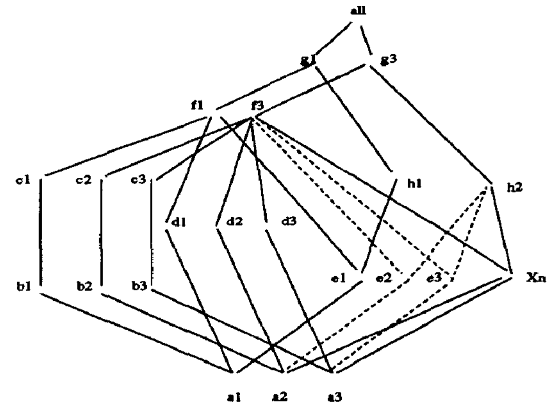

Figure 4: Merge operator

Update We also need an operator 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 reflected by a sequence of individual insertions and deletions.

Operator 4 (Update) *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* $x_n \notin instset(l_a)$; Update(d, l_a, x_a, x_n) *is a new dimension* $((dname, L, A, \preceq, \gg), RUP', DESC)$ *such that:* $RUP'^{l_j}_{l_a} = RUP^{l_j}_{l_a} \setminus \{(x_a, x_j) | RUP^{l_j}_{l_a}(x_a) = x_j\} \cup \{(x_n, x_j) | RUP^{l_j}_{l_a}(x_a) = x_j\}$; $RUP'^{l_a}_{l_j} = RUP^{l_a}_{l_j} \setminus \{(x_j, x_a) | RUP^{l_a}_{l_j}(x_j) = x_a\} \cup \{(x_j, x_n) | RUP^{l_a}_{l_j}(x_j) = x_a\}$; $RUP'^{l_j}_{l_i} = RUP^{l_j}_{l_i}$, *for all other levels* l_i, l_j.

3 Computing the Update Operators

In this section we discuss the implementation of our update operators in a relational representation. We do this at an abstract level that suggests a concrete implementation using e.g. SQL. 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. We present the details of the implementation of the *Reclassify* operators. We refer the reader to the full version of this paper [4] for details on the remaining operators.

3.1 Mapping Dimensions to Relations

Denormalized Representation The idea in the denormalized representation is to build a single table containing all the rollups in the dimension.

Schema The schema $S_d = (rname, A, \mathcal{F})$ of the relation is defined as follows: *rname* is *dname*; A contains an attribute l for each level $l \in L$; $\mathcal{F}$ contains a functional dependency $rname : l_a \to l_b$ 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 below in the dimension instance. Formally, $Leafs(l) = InstSet(l) \backslash (ran(RUP^l_{l_1}) \cup \ldots ran(RUP^l_{l_n}))$, 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 3 *Consider the dimension of Figure 2; its denormalized relation representation 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_1	c_2	e_2	f_2	$g1$
i_3	k_1	a_2	b_1	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	$g1$

Normalized Representation In this paragraph we propose a mapping from the multidimensional model to a normalized relational model, in order to perform the comparison between both approaches wrt dimension updates. The idea in the normalized representation is to build a table for each direct rollup in the dimension.

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 relation schema $name(l_i, All)[l_i]$ in schema Sc, where $name(l_i, All)$ 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 relation 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 $RUP^{l_j}_{l_i}$ as tuples; for each pair of levels $l_i, All \in L$ such that $l_i \preceq All$, we hav e a relation denoted as $R_{l_i All} = (S_{l_i All}, T_{l_i})$ with schema $S_{l_i All} = (name(l_i All), \{l_i\}, \{\phi\})$, and tuples $T_{l_i} = \{(e) \mid e \in Instset(l_i)\}$.

3.2 Primitive Operators

Structural Update Operators The implementation of the structural update operators is straightforward. The algorithm performs four types 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 rollup function includes the instance set of the level being generalized. For the *Relate* operator, these 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) of the updated schema. They have the same cost for all the operators. (d) Operations that update the instance, in general, additions/deletions of rollup 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 rollup function is added, the algorithm performs a join between the new rollup function and the dimension table; in the case of the normalized representation, the algorithm only adds the new rollup functions to the set of relations.

From the above paragraph we conclude that, in general, the execution of the *Generalize, Specialize* and *DelLevel* operators is 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 performs only operations of type (a) and (b), it has the same cost for both representations.

Instance Update Operators The algorithms for the instance update operators have two types 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 the execution costs of the algorithm in both representations requires 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) consist mainly in verifying that the inserted element reaches the same elements in the levels above it in the schema, through any possible path; for the *DelInstance* operator, operations of type (a) mainly consist in verifying that the element deleted is a leaf. In general, these operations will 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. A comparison between the costs of the operations of type (b) for both representations does not give a definitive result favoring either of the alternatives, in the general case.

3.3 Complex Operators

The procedural implementation of the update operators requires the execution of a set of standard operations over a DAG which represents the set of functional dependencies in the mapping; this DAG is the same in both representations. 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 by 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 over 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 also use sentences of the form *precondition:condition* meaning that the algorithm halts and does not perform the update if *condition* is false.

The algorithms for the *Reclassify* operator in the denormalized and normalized representations are depicted in Figure 5 and Figure 6, respectively. First, we verify 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. After this, the main task consists in verifying the precondition introduced in Theorem 1. This is done as follows: the procedure *FindConfLevels* finds a subset of the conflicting levels which contains all the minimal conflicting levels; it stores them in a variable $ConfLevels$. Then the minimal conflicting levels are computed and stored in the variable LTC (Levels To check). In the case of the denormalized representation, the recursive procedure $FindConfLevels$ computes the variable $ConfLevels$, traversing the subgraph below l_a. The vertices which are candidates to be added to $ConfLevels$ are stored in A_1, and are determined by the operation $Join(l_i, l_b, S_1)$ which computes the vertices reached by both l_i and l_b in S_1, which are not pairwise reachable. The relation R is computed to determine if there is a value in the instance set of l_i which reaches x_a. If this is the case, the levels in A_1 are added to $ConfLevels$. The procedure $FindConfLevels$ performs the minimum number of outer-joins to build the relation R: the set of levels to visit is pruned 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.

In the case of the denormalized representation, the procedure *ReachTuple* computes the tuples t_1 and t_2 which have the values reached by x_a and x_b, respectively, then it is verified that the tuples t_1 and t_2 have the same value in the attributes in $ConfLevels$. While in the denormalized representation *ReachTuple* is computed by a single selection, 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. However, 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 LTC. After computing LTC, the path from l_a to every every level l_i in LTC is computed. This path is required by the procedure *ReachTuple*, which performs the traversal of the path making a sequence of selections 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 the update over the relation when the precondition is computed is straightforward.

Main
precondition: there is an edge (l_a, l_b) in S_d
precondition: $InInstSet(l_b, x_b)$
$t_1 := ReachTuple(l_a, x_a)$
$t_2 := ReachTuple(l_b, x_b)$
Call *FindConfLevels*
$LTC := MinSet(ConfLevels, S_d)$
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 $ConfLevels := ConfLevels \cup A_1$

Figure 5: Algorithm for *Reclassify*$(R_d, l_a, l_b, x_a, x_b)$ in the denormalized representation.

Example 4 *Let us apply the algorithm of Figure 5 to the dimension depicted in Figure 2. Its denormalized representation is depicted in Example 3. Suppose we would like to reclassify a_2 from b_1 to b_2. After the first validations, we compute Reachtuple(A, a_2) and Reachtuple(B, b_1), obtaining:*
$t_1 = \langle null, null, a_2, b_1, c_1, e_2, f_2, g1\rangle$,
$t_2 = \langle null, null, null, b_2, c_2, e_1, f_1, g1\rangle$. *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.

To save 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 by deleting the edge A, B from S. G_1 will be the graph s.t. $V = \{A, I\}$, and $E = \{I - A\}$, that is, the graph whose edges are all the edges reaching A; in this example, only $I - A$. Applying the procedure LevelsToCheck(A) will

Main
precondition: There is an edge (l_a, l_b) in S_d
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 $PreSet(l_1, G_1) \setminus Visited$
  $A_1 := Join(l_i, l_b, S_1)$
  If $A_1 \not\subseteq ConfLevels$
  $R := R \; outerjoin \; R_{l_i l_1}$
  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 6: Algorithm for $Reclassify(R_d, l_a, l_b, x_a, x_b)$ in the normalized representation.

require only one recursive call, with argument I. Eventually, we get $LTC = \{C\}$. The ReachTuple procedure will have $\tau = \langle BC \rangle$ as one of its arguments. Thus, two tuples will be computed:
$t_1 = ReachTuple(\mathcal{R}_d, \langle BC \rangle, B = b_2)$,
$t_2 = ReachTuple(\mathcal{R}_d, \langle BC \rangle, B = R_{AB}(a_2))$. *The tuple $t_1 = \langle b_2, c_2 \rangle$ is obtained directly from R_{BC}, while $t_2 = \langle a_2, b_1, c_1 \rangle$ is obtained by 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.*

4 Conclusion and future work

Research on data warehouse evolution has focused on fact tables. However, dimensions are also subjected to evolve over time. In a former work [5] we presented a multidimensional model which accounts for dimension updates in data warehouses, including a set of primitive operators for performing such updates. In this paper we have extended the model with a set of semantically meaningful operators: *reclassification, merge, split,* and *update.* Although these operators could be implemented by transactions of basic operators, they are more efficiently performed as single operations. We also formally defined the conditions under which these operators preserve the consistency properties of a dimension.

We presented a mapping from the dimensional to the relational model, and its properties, using two approaches: denormalized and normalized relational tables (Star or Snowflake schemas, respectively [7]). We compared these approaches with respect to dimension updates, and showed the advantages of the denormalized approach.

Although several previous works on modeling OLAP existed [1, 3], only [2] and [8] deal formally with dimension modeling, but without considering dimension updates.

We are currently working on extending our operators in several ways. First, 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 intensionally, over elements satisfying some condition, and to express the modified rollup functions also intensionally.

Acknowledgments This work was partially supported by the Institute of Robotics and Intelligent Systems, and by the University of Buenos Aires.

References

[1] R Agrawal, A Gupta, and S. Sarawagi. Modeling multidimensional databases. *Proceedings of the 13th IEEE-ICDE Conference*, 1997.

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

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

[4] C. Hurtado, A. Mendelzon, and A. Vaisman. Updating OLAP dimensions (extended version). *URL: ftp://ftp.db.toronto.edu/pub/ papers/fulldolap99.ps.gz.*

[5] C. Hurtado, A. Mendelzon, and A. Vaisman. Maintaining data cubes under dimension updates. In *Proceedings of the IEEE-ICDE Conference*, pages 346–355, 1999.

[6] 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.

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

[8] W. Lehner. Modeling large scale OLAP scenarios. In *Proceedings of 6th International Conference on Extending Database Technology (EDBT'98)*, Valencia, Spain, November 1998.

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

[10] J Widom. Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM)*, pages 25-30, 1995.