Maintaining Data Cubes under Dimension Updates

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

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

Alejandro A. Vaisman
Universidad de Buenos Aires
av2r@dc.uba.ar

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. The usual assumption is that the data points reflect the dynamic aspect of the data warehouse, while dimensions are relatively static. However, in practice, dimension updates are often necessary to adapt the multidimensional database to changing requirements. Structural updates can also take place, like addition of categories or modification of the hierarchical structure. When these updates are performed, the materialized aggregate views that are typically stored in OLAP systems must be efficiently maintained. These updates are poorly supported (or not supported at all) in current commercial systems, and have received little attention in the research literature. We present a formal model of dimension updates in a multidimensional model, a collection of primitive operators to perform them, and a study of the effect of these updates on a class of materialized views, giving an algorithm to efficiently maintain them.

1 Introduction

A data warehouse is a large collection of integrated data, built to assist the decision-making process. The class of queries required on data warehouses for data analysis and visualization is known as OLAP, for online analytical processing. OLAP systems often support a multidimensional data model, according to which a data fact is viewed as a mapping from a point in a space of dimensions into one or more spaces of measures. For example, suppose we have data about policies sold by an insurance company. We may have different kinds of measures, such as number of policies sold, dollar amount, coverage amount, which can be analyzed in terms of dimensions such as kind of coverage, customer, salesperson, date of sale, geographic region, etc. Dimensions are often organized into hierarchical levels; for example, a geographic region could consist of a town or city name at the lowest level, with counties and then states or provinces and then areas at higher levels. These levels enable the definition of different levels of data aggregation, a central issue in data analysis.

In a relational implementation of OLAP, we can think of facts as being stored in fact tables, while each dimension is described in a dimension table. The usual assumption in OLAP systems is that fact tables reflect the dynamic aspect of the data warehouse, while data in dimension tables are relatively static. However, changes in the data will often require updates to the dimension tables: new salespersons may be hired or fired, new kinds of coverage introduced or discontinued, etc. In addition, structural changes to the dimension hierarchies also occur: for example, regions may be reorganized or merged or split. Kimball [11] talks about slowly changing dimensions, covering only the first kind of change discussed above, not structural changes.

An OLAP database usually stores materialized views which precompute aggregates in order to speed-up the complex queries often needed. This implies that, whenever an update is performed over the underlying database, the corresponding changes should also be applied to these views, which constitutes the well-known problems of view maintenance and incremental view maintenance [13, 6, 14].

In this paper we address the dimension update problem. Its main contributions are: (a) a formal model of dimension updates in a multidimensional model, covering updates to the domains of the dimensions and structural updates to the dimension hierarchies, together with a definition of a collection of primitive operators to perform these updates, and (b) a study of the effect of these updates over a class of materialized views over the dimension levels, and an algorithm to efficiently maintain them, which turns out to be more efficient than the well-known summary-delta method, when dealing with dimension updates.
The rest of the paper is organized as follows: In Section 2 we introduce our data model, over which we will describe dimension updates. In Section 3 we define dimension updates and the basic set of operators to perform them. In Section 4 we approach the problem of cube maintenance. We conclude in Section 5, presenting related work, conclusions, and comments on future directions.

2 Multidimensional Model

2.1 Dimensions and Fact Tables

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

**Definition 1 (Dimension Schema and Instance)**

A *dimension schema* is a tuple $(dname, L, \preceq)$, where $dname \in D$ is the name of the dimension, $L \subseteq L$ is a finite set of levels, which contains distinguished level name All, such that $dom(All) = \{all\}$. Finally, $\preceq$ is a relation over levels, such that $\preceq^*$, its transitive and reflexive closure, is a partial order, with a unique bottom level, called $l_{m,1}$, a unique top level, All, and, for every level $l \in L$, $l_{m,1} \preceq l$ and $l \preceq All$ hold. Moreover, if $l_a$ and $l_b$ are levels in $L$, and $l_a \preceq^* l_b$, then $l_a \preceq l_b$.

A *dimension instance* is a tuple $(D, RUP)$, where $D$ is a dimension schema, and $RUP$ is a set of partial functions such that (a) for each pair of levels $l_1, l_2$ such that $l_1 \preceq l_2$, there exists a roll-up function (partial function) $RUP^b_{l_1} : \text{dom}(l_1) \rightarrow \text{dom}(l_2)$, (b) 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 $\tau_2 = (l_1, l_2, \ldots, l_{m-1}, l_m)$, we have $RUP^b_{l_1} \circ \ldots \circ RUP^b_{l_{n-1}} = RUP^b_{l_1} \circ \ldots \circ RUP^b_{l_{m-1}}$, and (c) for each triple of levels $l_1, l_2$, $l_3 \in L$ such that $l_1 \preceq l_2$ and $l_2 \preceq l_3$, $\text{ran}(RUP^b_{l_1}) \subseteq \text{dom}(RUP^b_{l_3})$.

**Example 1** Figure 1 shows an example of a dimension schema and instance for a dimension Product. Here, for the dimension schema, $dname = Product$, $L = \{ \text{Item Id}, \text{Brand}, \text{Company}, \text{Category}, \text{Corporation}, \text{Product}, \text{All} \}$. Relation $\preceq$ contains the following pairs: $\text{Item Id} \preceq \text{Brand}, \text{Brand} \preceq \text{Company}, \text{Company} \preceq \text{Category} \preceq \text{Corporation} \preceq \text{All}$. 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 $l_m = l_n$, $RUP^*_{l_m} = \text{identity}$; 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 $\preceq$, $RUP^*_{l_m} = RUP^*_{l_m} \circ \ldots \circ RUP^*_{l_{n-1}}$.

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 $l \preceq l''$, $\text{dom}(RUP^*_{l}) = \text{dom}(RUP^*_{l''})$. Given a dimension and a pair of levels $l$ and $l'$, such that $l \preceq l'$, the *instance set of l*, or $\text{instset}(l)$, is the set containing values for the level, defined by $\text{instset}(l) = \text{dom}(RUP^*_{l})$. As an example, for the dimension instance $Product$ given in example 1 and Figure 1, the instance set of level $\text{Item Id}$, is $\{i_1, i_2, i_3, i_4\}$.

**Definition 2 (Fact Table)** A *fact table schema* is a tuple $s = (fname, Lset, m)$, where $fname \in F$ is a fact table name, $Lset$ is a set of levels, and $m$ is a level, called the measure of the fact table. Moreover, given a fact table schema $(fname, Lset, m)$, a point is a mapping from each level in $Lset$, to a value in $\text{dom}(l_i)$.

**Definition 3 (Multidimensional Database)**

A *multidimensional database schema* is a pair $MS = (DS, FS)$, where $DS$ is a set of dimension schemas, and $FS$ is a set of fact table schemas. A *multidimensional database instance* is a tuple $MD = (D, F)$, where $D$ is a set of dimensions, and $F$ a set of fact tables.

**Notation** Given a dimension set $D$, a *level group* is a set of levels containing exactly one level for each di-
mension in \(D \setminus \{\text{Measure}\}\), where \(\text{Measure}\) is a distinguished dimension (see below). \(GB_D\) will denote the set of all the possible level groups, and \(GBottom_D\) a level group containing the bottom levels of each dimension in \(D\). We also define a base fact table as a fact table with schema \((fname, GBottom_D, m)\).

2.2 Data Cubes

Several classes of aggregate views have been used to fulfill different requirements in OLAP systems. Gray et al [5] introduced the data cube operator as a shorthand for a set of cube views that contains data from a base fact table, aggregated over all the possible groups of attributes in it. We will extend it in order to to include views computing aggregates over the levels of the dimensions, and define the operator \(CubeView\) to express them. We also assume that every set of dimensions contains a distinguished one, called \(\text{Measure}\), with level \(m\), over which aggregation takes place, and define an aggregate function \(Ag\) over a level \(m\) as a function having signature \(Ag : \text{dom}(m) \rightarrow \text{dom}(m)\).

Definition 4 (Cube View) Given a set of dimensions \(D\), a base fact table \(f_{base}\), with measure \(m\), and a level group \(GB = \{l_1, ..., l_n\}\); \(CubeView(f_{base}, D, GB)\) yields a fact table \(f\), with schema \(s = (fname, GB, m)\), and instance defined as follows: given a point \(c\), \(S(c)\) is the set that contains points \(c'\) in the domain of \(f_{base}\), such that for each pair of levels \(l_1 \in GBottom_D\) and \(l_2 \in GB\), belonging to the same dimension, we have \(c(l_2) = RUP^{l_2}_{l_1}(c'(l_1))\), where \(RUP^{l_2}_{l_1}\) is in the set of roll up functions \(RUP\) of that dimension. Then, for the points \(c\) such that \(S(c) \neq \emptyset\), we have \(f(c) = Ag(S(c))\). Notice that \(m\) and \(Ag\) are parameters of \(CubeView\), because they can be inferred from the context.

Example 2 Consider a set of dimensions \(D\)={Product,Store,Time,Sales}. The schema and roll up functions for Product are given in figure 1(a). Schemas for Time and Store can be inferred from the rollup functions given below. For Store we have \(RUP_{\text{storeId}}\) = \{\(s_1 \mapsto t_1, s_2 \mapsto t_2, s_3 \mapsto t_3\)\}; for Time we have: \(RUP_{\text{day}}\) = \{\(d_1 \mapsto w_1, d_2 \mapsto w_2, d_3 \mapsto w_3\)\} (we have omitted the roll ups to All). Let us denote the following fact table as DailySales:

<table>
<thead>
<tr>
<th>ItemId</th>
<th>StoreId</th>
<th>Day</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_1)</td>
<td>(s_1)</td>
<td>(d_1)</td>
<td>10</td>
</tr>
<tr>
<td>(i_2)</td>
<td>(s_1)</td>
<td>(d_1)</td>
<td>20</td>
</tr>
<tr>
<td>(i_2)</td>
<td>(s_2)</td>
<td>(d_1)</td>
<td>20</td>
</tr>
<tr>
<td>(i_2)</td>
<td>(s_2)</td>
<td>(d_2)</td>
<td>40</td>
</tr>
<tr>
<td>(i_2)</td>
<td>(s_3)</td>
<td>(d_2)</td>
<td>30</td>
</tr>
</tbody>
</table>

Then, assuming that \(Ag\) is the SUM aggregate function, and the measure dimension is Sales (with the measure level also named Sales), the fact table \(CubeView(\text{DailySales}, D\{\text{ItemId, StoreId, Week}\})\), call it SalesJSW, is the following:

<table>
<thead>
<tr>
<th>ItemId</th>
<th>StoreId</th>
<th>Week</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_1)</td>
<td>(s_1)</td>
<td>(w_1)</td>
<td>10</td>
</tr>
<tr>
<td>(i_2)</td>
<td>(s_1)</td>
<td>(w_1)</td>
<td>20</td>
</tr>
<tr>
<td>(i_2)</td>
<td>(s_2)</td>
<td>(w_1)</td>
<td>60</td>
</tr>
<tr>
<td>(i_2)</td>
<td>(s_3)</td>
<td>(w_2)</td>
<td>30</td>
</tr>
</tbody>
</table>

Definition 5 (Data Cube Operator) Given a dimension set \(D\), a base fact table \(f_{base}\) with measure \(m\), and a set of level groups \(GBSET\), the data cube operator, \(DataCube(D, f_{base}, GBSET)\), gives a multidimensional database \((D, F \setminus f_{base})\), such that for every level group \(GB\) in \(GBSET\) there is one and only one fact table \(f\) in \(F\), where \(f = CubeView(f_{base}, D, GB)\).

Example 3 For the dimension set \(D\) and the base fact table \(DailySales\) of example 2, we want to materialize views over the following set of level groups: \(GBSET = \{\{\text{ItemId, StoreId, Day, Sales}\}, \{\text{ItemId, StoreId, Week, Sales}\}, \{\text{Brand, StoreId, Week, Sales}\}, \{\text{Brand, All, All, Sales}\}\}\). \(DataCube(D, f_{base}, GBSET)\) will contain a base fact table \(DailySales\), and four aggregate views, which represent the sum of the measure Sales, grouped by each level group in \(GBSET\).

2.3 ROLAP Storage of a Multidimensional Database

In this subsection we will define how the multidimensional database is stored using the relational model, and denote this representation as a ROLAP database. We will not get into the details due to space available. In ROLAP terminology, \(D\) will be a set of dimension tables, and \(F\), a set of fact tables (i.e., we will use the same names as in section 2.1, confusions are avoided by the context). The ROLAP database that stores \((D, F)\) will have a dimension table \(d \in D\) for each dimension \(d = ((fname, L, \leq), RUP) \in D\), with name \(dname\). The set of levels in \(d\), excluding level \(All\), will be its attributes, each attribute denoted as \(A_t\). For every pair of levels, \(l_1, l_2\) in \(L\), where \(l_1 \leq l_2\), the functional dependency \(l_1 \rightarrow l_2\) holds. Its instances will be such that for each \(c \in instset(l_{i=1})\) there exists a tuple \(t\), s.t. \(t.A_{i=1} = c\), and if \(c' \in instset(l)\), \(RUP^l_{l_{i=1}}(c) = c'\), then \(t.A_i = c'.i\). Moreover, for every fact table \(f\) in \(F\), with schema \((fname, Lset, m)\), we will have a relation \(f\), with schema name \(fname\), and attributes \(Lset \cup \{m\}\). The functional dependency \(Lset \rightarrow m\) holds. The fact table instance contains the tuples that represent the graph of the fact table.
Example 4 Dimension Product in figure 1, is mapped to the table shown below. The functional dependencies \( \text{ItemId} \rightarrow \text{Brand}, \text{Brand} \rightarrow \text{Company}, \text{Company} \rightarrow \text{Corporation}, \text{ItemId} \rightarrow \text{category}, \text{and Category} \rightarrow \text{Corporation} \) hold.

<table>
<thead>
<tr>
<th>ItemId</th>
<th>Brand</th>
<th>Company</th>
<th>Category</th>
<th>Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_1</td>
<td>b_1</td>
<td>c_0</td>
<td>c_1</td>
<td>c_1 r_1</td>
</tr>
<tr>
<td>t_2</td>
<td>b_2</td>
<td>c_0</td>
<td>c_1</td>
<td>c_1 r_1</td>
</tr>
<tr>
<td>t_3</td>
<td>b_2</td>
<td>c_0</td>
<td>c_1</td>
<td>c_1 r_1</td>
</tr>
<tr>
<td>t_4</td>
<td>b_2</td>
<td>c_0</td>
<td>c_2</td>
<td>c_1 r_2</td>
</tr>
</tbody>
</table>

3 Dimension Updates

We will define a basic set of operators that will allow us to modify either the schema or an instance of a given dimension, classifying them into two subsets: Structural Update Operators and Instance Update Operators. Operators in the first set modify the structure of a dimension, while operators in the second one do the same with an instance of a dimension.

3.1 Structural Update Operators

The Generalize operator creates a new level, \( l_n \), to which a pre-existing one, \( l \), rolls up. A function \( f \) must be defined from the instance set of \( l \), to the domain of \( l_n \).

Operator 1 (Generalize) Given a dimension \( d = ((\text{name}, L, \leq), \text{RUP}) \), two levels \( l \in L, l_n \not\in L \), and a function \( f_i^k : \text{instset}(l) \rightarrow \text{dom}(l_n) \), Generalize\((d, l, l_n, f_i^k)\) is a new dimension \( ((\text{name}, L \cup \{l_n\}, \leq \cup \{(l, l_n), (l_n, \text{All})\} \setminus \{(l, \text{All})\}, \text{RUP}^g) \), where \( \text{RUP}^g \) is the set containing the roll up functions \( \text{RUP}^g_{l_n} \), such that:

\[
\text{RUP}^g_{l_n} = f_i^k ; \text{RUP}^g_{\text{All}} = \{(e, \text{all}) \mid e \in \text{ran}(f_i^k)\} ; \text{RUP}^g_{l_n} = \text{RUP}^g_{l_i} \text{ for all other levels } l_i, l_j.
\]

The Specialize operator adds a new level \( l_n \) to a dimension. Level \( l_n \) will roll up to the lowest level of \( L, l_{inf} \), becoming the lowest level of the dimension. Again, a function must be defined for this roll up.

Operator 2 (Specialize) Given a dimension \( d = ((\text{name}, L, \leq), \text{RUP}) \), a level \( l_n \not\in L \), and a function \( f_i^{inf} : \text{dom}(l_n) \rightarrow \text{instset}(l_{inf}) \), Specialize\((d, l_n, f_i^{inf})\) is a new dimension of the form \( ((\text{name}, L \cup \{l_n\}, \leq \cup \{(l_n, \text{inf})\}), \text{RUP}^s) \), where \( \text{RUP}^s \) is the set containing the roll up functions \( \text{RUP}^s_{l_n} \), such that \( \text{RUP}^s_{l_n} = f_i^{inf} \), and \( \text{RUP}^s_{l_i} = \text{RUP}^s_{l_i} \text{ for all other levels } l_i, l_j.

Example 5 Suppose we want to define a new level in the Store dimension, say, the type of store, with two possible values, \( t_1 \) and \( t_2 \). The operation would be defined as: Generalize(Store, StoreId, Type, \( f_{\text{StoreId}}^{\text{Type}} \)), where \( f_{\text{StoreId}}^{\text{Type}} = \{(s_1, t_1), (s_2, t_1), (s_3, t_2)\} \). See figure 2(a). Then, we specialize the Time dimension. The only level at which we can do this, is Day. The operator will be defined as: Specialize(Time, Hour, \( f_{\text{Hour}}^{\text{Day}} \)). Figure 2(b) shows a limited portion of the dimension instance after specialization.

Before we introduce the next operator, let us define the following notation: we say that two levels are independent, denoted \( l_a \not\equiv \text{all } l_b \), when \( l_a \not\equiv \text{all } l_b \) and \( l_b \not\equiv \text{all } l_a \).

The Relate operator defines a roll up function between two independent levels belong to a dimension. 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. Otherwise, the Relate cannot be applied. Moreover, when conditions for relating two levels \( l_a \) and \( l_b \) are met, we must delete all the redundant roll up functions that may appear, which are not admitted in our model (we only include direct roll ups in it). For instance, we must delete the roll up functions between levels \( l_a \) and \( l_b \) such that \( l \not\equiv \text{all } l_a \) and \( l \not\equiv \text{all } l_b \).

Operator 3 (Relate Levels) Given a dimension, say \( d = ((\text{name}, L, \leq), \text{RUP}) \), a pair of levels \( l_a \in L, l_b \in L \), such that: \( l_a \not\equiv \text{all } l_b \), and the existence of a function \( f_{l_a}^{l_b} \) between \( \text{instset}(l_a) \) and \( \text{instset}(l_b) \), defined as \( f_{l_a}^{l_b} = \{i_a \mapsto i_b \mid \exists i \in l_{inf}, \text{RUP}^s_{l_a} \cap \text{RUP}^s_{l_b}(i) = i_a, \text{RUP}^s_{l_{inf}}(i) = i_b\} \); Relate\((d, l_a, l_b)\) is the dimension \( ((\text{name}, L, \leq), \text{RUP}^r) \), where \( \text{RUP}^r \) is the set containing the roll up functions \( \text{RUP}^r_{l_a} \), \( \text{RUP}^r_{l_b} \), and \( \text{RUP}^r_{l_i} = \text{RUP}^r_{l_i} \text{ for all other levels } l_i, l_j \).

Note that we do not need to specify \( f_{l_a}^{l_b} \) in the definition of the operator, because it is uniquely determined by

![Figure 2](image-url)

(a) Store dimension after Generalization  (b) Time dimension after Specialization.
the pre-existent roll up functions.

The Unrelate operator deletes a relation \( \leq \) between two levels \( l_a \) and \( l_b \), such that \( l_a \leq l_b \). 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 \( l_a \leq l_b \) and \( l_b \leq l_c \), we must preserve \( l_a \leq l_c \) by making it explicit, in case we delete \( l_a \leq l_b \), because, again, this relation was only implicit in the model.

**Operator 4 (Unrelate Levels)** Given a dimension \( d = ((\text{dname}, L, \leq), \text{RUP}) \), and two levels, \( l_a \in L \) and \( l_b \in L \), such that \( l_a \leq l_b \), Unrelate\( (d, l_a, l_b) \) is a new dimension \( ((\text{dname}, L', \leq'), \text{RUP}') \), where \( \leq' = \leq \setminus \{(l_a, l_b) \} \cup \{(l_i, l_j) \mid l_i \leq l_j \wedge l_a \leq l_b \} \cup \{(l_i, l_j) \mid l_i \leq l_j \wedge l_a \leq b \} \), \( \leq_{ab} = \leq \setminus \{(l_a, l_b) \} \), and \( \text{RUP}'_{i_a} = \text{RUP}'_{i_a} \circ \text{RUP}'_{i_b} \) if \( l_i \leq l_j \), \( \text{RUP}'_{i_a} = \text{RUP}'_{i_a} \circ \text{RUP}'_{i_b} \), if \( l_i \leq l_a \); \( \text{RUP}'_{i_a} = \text{RUP}'_{i_i} \), for all other levels \( l_i, l_j \).

**Example 6** Operation Relate(Product, Brand, Category) will relate level Brand with level Category (figure 3). Notice that relation ItemId \( \leq \) Category was deleted as a collateral action. After this, we unrelate levels Category and Corporation by means of Unrelate(Product, Category, Corporation). We must add to \( \leq \), the relation Category \( \leq \) All. Otherwise, level Category would not be able to reach level All. Note also that Brand \( \leq \) All is not required because All is reached from Brand in \( \leq \) Category Corporation.

The DelLevel operator deletes a level and its roll up functions. The level to be deleted cannot be the lowest one in the dimension \( (l_{i_0}, l_f) \), unless it rolls up to only one higher level(for instance, we could delete level "hour" in the schema corresponding to the instance of

\[ \text{figure 2(b), but we could not delete level ItemId in figure 1.} \] As it was the case with the Relate operator, taking into account that we only define the direct roll ups, when deleting a level we must add the functions between levels above and below it.

**Operator 5 (Delete Level)** Given a dimension \( d = ((\text{dname}, L, \leq), \text{RUP}) \) and a level \( l \in L \), \( l \neq \text{All} \), such that, if \( l = l_{i_f} \), then there is only one level \( l_j \) such that \( l \leq l_j \), DelLevel\( (d, l) \) is a new dimension \( ((\text{dname}, L' = L \setminus \{l\}, \leq'), \text{RUP}') \), where \( \leq' = \leq \setminus \{(l_1, l_2) \mid (l_1 = l) \vee (l_2 = l) \} \cup \{(l_1, l_2) \mid (l_1 \leq l) \wedge (l_2 \leq l) \} \setminus \{l \} \), and \( \text{RUP}'_{i_1} = \text{RUP}'_{i_1} \circ \text{RUP}'_{i_2} \) if \( l_i \leq l_j \), \( \text{RUP}'_{i_1} = \text{RUP}'_{i_1} \circ \text{RUP}'_{i_2} \), if \( l_i \leq l_a \); \( \text{RUP}'_{i_1} = \text{RUP}'_{i_1} \), for all other levels \( l_i, l_j \).

**Example 7** Let us suppose we want to delete level Brand from the Product dimension. We would obtain the schema and instance depicted in figure 4.

### 3.2 Instance Update Operators

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

The AddInstance operator inserts a new element, say \( x \), into a level \( l_a \) (i.e., an element not belonging to the instance set of \( l_a \)). We must provide the operator with the pairs \( (l_i, x_i) \), such that every \( l_i \) is a level to which \( l_a \) directly rolls up \( (l_a \leq l_i) \), and \( \text{RUP}'_{i_a}(x) = x \).

**Operator 6 (Add Instance)**

Given a dimension \( d = ((\text{dname}, L, \leq), \text{RUP}) \), a level \( l_a \), an element \( x_a \in \text{dom}(l_a), x_a \notin \text{instset}(l_a) \), and a set of pairs \( P = \{(l_1, x_1), \ldots, (l_n, x_n)\} \), where \( l_k \) is a level, and \( x_k \) is an element, and(a) for each \( (l_k, x_k) \in P \), \( x_k \in \text{instset}(l_k), (b) \text{dom}(P) = \{l_a \mid l_a \leq \} \)
Figure 5: (a) Add instance (b) Delete instance.

l_{k}\}, and (c) for each pair (l_{k}, z_{k}), (l_{s}, x_{s}) \in P such that exists a level l \in L, l_{k} \leq l and l_{s} \leq l : RUP'_{l_{k}}(z_{k}) = RUP'_{l_{s}}(x_{s}), AddInstance(d, l, x_{a}, P) is a new dimension \(((dname, L, \leq), RUP')\), where RUP' is the set containing the roll up functions RUP'^{i_{j}}, such that RUP'^{i_{j}} = RUP'^{i_{j}} \cup \{(x_{a}, z)\}, if (l_{j}, z) \in P; RUP'^{i_{l_{i}}} = RUP'^{i_{l_{i}}} for all other levels l_{i}, l_{j}.

The DelInstance operator deletes an element belonging to the instance set of a level l_{a}. It is only defined when no element of any level l_{i}, such that l_{i} \leq l_{a}, rolls up to the element being deleted.

Operator 7 (Delete Instance) Given a dimension \(d = ((dname, L, \leq), RUP)\), a level l_{a} \in L, and an element x_{a} \in instset(l_{a}), \(x_{a} \notin \bigcup_{l \in L} ran(RUP'^{l_{a}})\), DelInstance(d, l_{a}, x_{a}) is the dimension \(((dname, L, \leq), RUP^{d})\) where RUP^{d} is the set containing the roll up functions RUP'^{i_{j}}, such that RUP'^{i_{j}} = RUP'^{i_{j}} \setminus \{(x_{a}, RUP'^{i_{j}}(x_{a}))\}; RUP'^{i_{l_{i}}} = RUP'^{i_{l_{i}}} for all other levels l_{i}, l_{j}.

Example 8 Suppose we want to add a new item, say i_{3}, to an instance of the Product dimension, by means of: AddInstance (Product, ItemId, i_{3}, \{(Brand, b_{3}), Category, c_{2}\}). Figure 5(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 5(b).

4 Maintenance

In this section we treat the evolution of the data cube under dimension updates, addressing separately structural and instance updates. Due to space limitations, all proofs will be omitted.

In order to specify the maintenance algorithms, we will use the relational algebra with bag semantics, extended with the generalized projection operator to express aggregation([7]). The generalized projection operator will be denoted as \(\Pi_{A}\), where the projected attributes can include aggregate functions. Then, computing CubeView\((D, f_{\text{base}}, GB)\) is equivalent to computing the relation: \(\Pi_{GB \setminus \{All\}, Ag(m)} f_{\text{base}} \Join d_{1} \ldots \Join d_{n}\), where \(d_{1}, \ldots, d_{n}\) are the dimensions of the levels that are in \(GB \setminus \{All\}\) but not in \(G\text{Bottom}_D\).

For notation, when using the generalized projection, we will refer to \(GB \setminus \{All\}\) just as \(GB\). In this way, the cube view of example 2 can be expressed as: 
\(\Pi_{\text{ItemId, StoreId, Week, Sum(S)Sales}} \Join \text{Time}\)

4.1 Structural Updates

When a dimension structural update occurs, we must guarantee that the new data cube represents the aggregation of the base fact table in the new dimension set. However, when a bottom level of some dimension is deleted, or the dimension is specialized, the base fact table of the new set is no longer the same as the base fact table of the original set.

To handle this problem, we propose the following data cube adaptation: if a DelLevel of a bottom is specified, recompute \(f_{\text{base}}\) as the cube view of the old base fact table grouped by the bottom level group of the new dimension set \(D'\). If the update is Specialize, \(f_{\text{base}}\) will not longer be a base fact table of \(D'\), and the new base fact table over \(D'\) of which \(f_{\text{base}}\) is a cube view is not uniquely determined. We will not address in this paper the problem of finding an appropriate new base fact table, but it can be seen that one always exists.

The determination of a new base fact table is related to the reconstruction problem [4], which consists on the estimation and computation of a fact table based on an aggregate view of it.

We can summarize our simple incremental maintenance algorithm for structural updates as follows. For the operators Relate, UnRelate, and Generalize, no maintenance is required. If the update is DelLevel\((l, i)\) : (a) if \(l \in G\text{Bottom}_D\), compute a new base fact table as \(\Pi_{G\text{Bottom}_D, Ag(m)} f_{\text{base}} \Join d_{l}\), perform data cube adaptation, if needed; (b) delete every table \(f\) in \(F\) s.t. \(l\) belongs to its schema. Finally, if the update is Specialize\((d, l_{0}, RUP'^{l_{0}})\), compute a fact table \(f'_{\text{base}}\) such that \(f_{\text{base}} = \Pi_{G\text{Bottom}_D, Ag(m)} f'_{\text{base}} \Join d_{l_{0}}\), and drop \(f_{\text{base}}\).
4.2 Instance Updates

Assuming a ROLAP storage of the data cube, an instance update reduces to the insertion and/or deletion of some tuples in the dimension tables. We could therefore apply existing incremental maintenance algorithms for materialized relational views with aggregates [1, 15, 14]. However, we will see that we can do better by exploiting the special form of these updates. In particular, we will show how to improve on the “summary delta” method of Munick et al. [14].

In general, incremental maintenance involves (1) computing the set of changes, sometimes called the delta table (propagation phase), and (2) applying the changes represented in the delta table to the materialized view (refresh phase). In the case of views with aggregations, this approach applies only if the aggregate functions are self maintainable [13]. We will assume in what follows that the aggregate operator $Ag$ is $SUM$, and that we extend the tables and the cube views to store the count value required to make $SUM$ self-maintainable with respect to insertions and deletions.

In what follows, we will use a reduced version of the data cube of Example 3.

Example 9 To show the ideas depicted in this section, we will be considering the set of dimensions $D$ = {Product, Store, Time, Sales}, with schemas specified as follows: Product has hierarchy: ItemId $\leq$ Brand $\leq$ All; Store has hierarchy: StoreId $\leq$ All; and Time has hierarchy: Day $\leq$ Week $\leq$ All. The roll up functions are the following (the roll ups to All are omitted, where possible): $RUP_{Brand}^{Brand} = \{i_1 \mapsto b_1, i_2 \mapsto b_2, i_3 \mapsto b_3\}$; $RUP_{StoreId}^{All} = \{s_1 \mapsto all, s_2 \mapsto all\}$; $RUP_{Week}^{Day} = \{d_1 \mapsto w_1, d_2 \mapsto w_1, d_3 \mapsto w_2\}$. We will use the same base fact table of example 2, called DailySales, so we will not repeat it here. The data cube is $dc = DataCube(D, f_{base}, GBSET)$, and each cube view will be denoted as “Sales” appended with the initials of the levels in the group level defining it (e.g. Sales_ASD represents the cube view where $GB = \{All, StoreId, Day\}$).

4.2.1 Maintaining Cube Views Independently

Under instance updates in a dimension, we can proceed as in the summary delta method, i.e., separately compute for each cube view a maintenance expression, derived using the rules presented by Quass [15]. The maintenance expression produces the change to each cube view, which is then applied to the cube view using a refresh algorithm. Let $Refresh(f, \Delta f)$ be the result of applying the refresh algorithm for set of changes $\Delta f$ to fact table $f$. Given a cube view $f = CubeView(D, f_{base}, GB)$, and an instance update $u$ that updates cube view $f$ to cube view $f' = CubeView(D, f_{base}, GB)$, we will denote by $\Delta f$ the change to the cube view $f$ wrt the update, that is, $\Delta f$ is a fact table such that $f' = Refresh(f, \Delta f)$.

Example 10 Consider the data cube of example 3, and an update DelInstance(Product, ItemId, i2) in the Product dimension, represented by $\Delta^- Product$. The maintenance expression for the cube view Sales$_{BSW}$ = CubeView(D, f$_{base}$, (Brand, Store, Week)), would be $\Delta Sales_{BSW}$ = $\Pi_{Brand, Store, Week}$ = $SUM$(Sales$_{BSW}$) $\Delta^- Product$ $\otimes$ Time. The new cube view, CubeView(D', DailySales, {Company, Store, Week}), is computed, in the refreshing stage, from Sales$_{BSW}$ and the delta change of $f$, using the refreshing algorithm, call it Refresh(f, $\Delta f$). Thus, Sales$_{BSW}' = Refresh(Sales_{BSW}, \Delta Sales_{BSW})$. This is done separately for every cube view in $dc$.

Now, we can say that if the instance update is over a level that is not the bottom level, any cube view change is empty. These updates are called irrelevant updates [8] in the view maintenance literature. Formally, given a dimension $d$, a dimension update, DelInstance(d, l, i) or AddInstance(d, l, i, P) is irrelevant (i.e., the dimension changes do not have an effect in any cube view) if $l$ is not the bottom level of $d$, or $\sigma_{l = i} f_{base} = \phi$. Thus, no maintenance is required in case the update is performed over a level that is not the bottom of the dimension. Furthermore, if the update is relevant, the delta changes that have an effect over the cube views are: $\Delta^+ d = \sigma_{l=i} d'$, in case of an AddInstance, and $\Delta^- d = \sigma_{l=i} d$, when a DelInstance takes place. Both of these contain exactly one tuple, which we will call $\delta^+ d$ and $\delta^- d$, respectively. In Figure 6 we give a set of rules to derive maintenance expressions for instance updates. Note that if rule 1 holds, no aggregation is needed.

Thus, the incremental maintenance algorithm can be stated as follows: check if the updates are irrelevant. If not, using the rules in figure 6, derive the maintenance expression to compute the changes for every cube view, and then compute the updated cube view using the refresh operator.

Example 11 Consider the retail data cube of example 9 and the update DelInstance(Product, ItemId, i2). Using rule 1, we can compute the delta change associated to a cube view DeltaSales$_{BSD}$, from the fact table DailySales (see example 2), with the following expression, avoiding any aggregate computation. The resulting table is also shown.
Given a instance update $u$, and a cube view $f = \text{CubeView}(D, \text{base}, GB)$

1. If $u = \text{DelInstance}(d, l, i)$, and $GBottom_D \setminus GB = \{l\}$, then:
$$\Delta f = \Pi_{GB \setminus \{l\}} f_{l} \cdot \Delta d_{l} \cdot d_{m} \cdot \sigma_{l_{m}} f_{\text{base}}$$
where $d_{l} = GB \setminus GBottom_{D}$
else
$$\Delta f = \Pi_{GB \setminus \{l\}} f_{l} \cdot \Delta d_{l} \cdot d_{m} \cdot \sigma_{l_{m}} f_{\text{base}} \bowtie d_{l} \bowtie \ldots \bowtie d_{n}$
where $l$, $l'$ is the level in GB that belongs to $d$, and $d_{l} \ldots d_{n}$ are the dimensions such that the levels in $GB \setminus \{d\} \setminus GBottom_{D}$, belong to.

2. If $u = \text{AddInstance}(d, l, i, P)$ then
$$\Delta f = \Pi_{GB \setminus \{l\}} f_{l} \cdot \Delta d_{l} \cdot d_{m} \cdot A_{d_{m}} \cdot \sigma_{l_{m}} f_{\text{base}} \bowtie d_{l} \bowtie \ldots \bowtie d_{n}$$
where $l$, $l'$ is the level in GB that belongs to $d$, and $d_{l} \ldots d_{n}$ are the dimensions such that the levels in $GB \setminus \{d\} \setminus GBottom_{D}$, belong to.

Figure 6: Maintenance Expressions for instance updates.

<table>
<thead>
<tr>
<th>$\Delta Sales_{BSD} = \Pi_{GB \setminus {d}} \cdot f_{\text{base}}$</th>
<th>$\Delta Sales_{DailySales}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brand</td>
<td>Store</td>
</tr>
<tr>
<td>$b_{2}$</td>
<td>$s_{1}$</td>
</tr>
<tr>
<td>$b_{2}$</td>
<td>$s_{2}$</td>
</tr>
<tr>
<td>$b_{2}$</td>
<td>$s_{2}$</td>
</tr>
</tbody>
</table>

**4.2.2 Maintenance Using the View Derived Lattice**

Because we have a set of related views, and, as we will see, the set of cube view changes are also related, we can take advantage of previous cube view changes to compute other ones which depend on them. We will present an efficient algorithm for the cases of instance updates, by adapting the view lattice introduced by Harinarayan et al. [10], and the summary delta method, to our specific needs.

The following data cube contains the cube view changes caused by our dimension update: Let $\Delta f_{\text{base}} = \Pi_{GBottom_{m}, \ldots, m} f_{\text{base}}$, if the update is $\text{DelInstance}(d, l, i)$, and $\Delta f_{\text{base}} = \sigma_{l_{m}} f_{\text{base}}$ if the update is $\text{AddInstance}(d, l, i, P)$. Then the delta cube, $\Delta c = \text{DataCube}(D \setminus \{d\} \cup \Delta d, \Delta f_{\text{base}}, GBSET)$, where $\Delta d = \delta d_{l} = \sigma_{l_{m}} d$ for $\text{DelInstance}$, and $\Delta d = \delta d_{l} = \sigma_{l_{m}} d'$ for $\text{AddInstance}$.

The data cube defined in this way contains the cube view changes to the data cube $(D, f_{\text{base}}, GBSET)$. We now present an algorithm to compute this cube efficiently. We assume that $GBSET = GB_{D}$, i.e., the data cube is totally materialized; the algorithm that we will propose can be extended to a partially materialized data cube.

We define the view lattice as follows: there is a node, denoted $N(GB)$, for each level group $GB$ in $GB_{D}$. The edges are defined as follows. If there is a 1-1 function $co$ between two level groups $GB_{1}$ and $GB_{2}$, such that for each level $l$ in $GB_{1}$, $l \leq co(l)$, then: there is an edge from $N(GB_{1})$ to $N(GB_{2})$, and this edge is labeled by the dimensions $d_{1}, \ldots, d_{n}$ of each level $l$ in $GB_{1}$ such that $l \neq co(l)$. We denote this edge by $N(GB_{1}) \preceq d_{1}, \ldots, d_{n} N(GB_{2})$. Finally, we associate with each edge $N(GB_{1}) \preceq d_{1}, \ldots, d_{n} N(GB_{2})$ a bag algebra expression to compute the view cube change $\Delta f_{2}$ associated to $GB_{2}$ from the cube view change $\Delta f_{1}$ associated to $GB_{1}$: $\Delta f_{2} = \Pi_{GB \setminus \{d\}} \Delta f_{1}$, $\bowtie d_{1} \bowtie \ldots \bowtie d_{n}$.

We now present a set of rules that, given an update to level $l$ of dimension $d$, modify the algebra expressions associated with the edges of the view lattice. We only consider “direct” edges, that is, those edges between sets $GB_{1}$ and $GB_{2}$ that differ only on one level.

**RULE A** For the direct edges $N(GB_{1}) \preceq d N(GB_{2})$ such that $l \in GB_{1}$, $GB_{1} \setminus GB_{2} = \{l\}$, $\Delta f_{2} = \Pi_{GB \setminus \{l\}} f_{l} \cdot \delta d_{l} = \delta d_{l}$, if the update is $\text{DelInstance}(d, l, i, P)$ or $\delta^{-} d$ if it is $\text{AddInstance}(d, l, i)$; $\Delta f_{2} = \delta d_{l} = \Delta f_{1}$.

**RULE B** For the direct edges $N(GB_{1}) \preceq d N(GB_{2})$ s.t. $l \notin GB_{1}$ and $GB_{1} \setminus GB_{2} = \{l\}$, $\Delta f_{2} = \Delta f_{1}$.

**RULE C** The remaining direct edges, of the form $N(GB_{1}) \preceq d_{1} N(GB_{2})$, are modified in the following way: For each level $l_{i} \in d$, consider the nodes $N(GB)$ such that $l_{i} \in GB$, and the edges between these nodes. These edges form a sub-lattice of the view lattice (see Figure 7). Let $\tau$ be a path from the bottom level of this sub-lattice s.t. $N(GB_{1}) \preceq d_{1} N(GB_{2}) \in \tau$. If there is an edge of the form $\preceq d_{1} \preceq d_{2} \in \tau$, then $\Delta f_{2} = \Pi_{GB \setminus A_{d_{1}} \setminus A_{d_{2}}} \Delta f_{1}$, where $A_{d_{2}}$ represents the attributes of the dimension $d_{2}$. If not (i.e., this is the first occurrence of $d_{1}$ in the path), then $\Delta f_{2} = \Pi_{GB \setminus A_{d_{1}}} \Delta f_{1} \bowtie d_{1}$.

**Example 12** An optimized lattice for our running example is depicted in figure 7. The dashed lines represent the derived expressions changed using rule A; the dotted lines represent the derived expressions changed using rule B; the solid lines represent the derived expressions changed with rule C. Here, the edges are labeled with a dimension name, when the join with the dimension is included in the expression.

The following algorithm generates a plan that leads to reduce the number of joins and aggregate computations while performing view maintenance due to instance updates.

**Algorithm 1**

**Input:** A data cube $dc$, its derived lattice, and an instance
update u.

Output: A plan (a subgraph of the optimized derived lattice), that chooses one direct predecessor for each node.

- For each node $N(GB)$ such that $l \in GB$:  
  case $u = \text{AddInstance}(d, l, i)$, the view change will be: $\Delta f = \Pi_{d, l_i \in \text{d}(l), i = 1}^{\text{d}(l_i), \max - \text{m} \sigma_l i f}$, where $l_i$ are the levels of $d$, $s.t. l_i \neq l$.  
  case $u = \text{AddInstance}(d, l, i, P)$, choose a node $GB'$, immediately below $GB$, using one of the well-known methods for computing aggregates (e.g. [2]). As a default, the predecessor with the least estimated size could be used ([2]).  
- For each node $N(GB)$ such that $l \notin GB$: choose as predecessor the node $N(GB')$, immediately below (i.e., connected by a direct edge), s.t. that $GB' \setminus GB = \{l'\}, l' \in d$.

The strength of Algorithm 1 resides in that only in the case of an AddInstance, and for the nodes $N(GB)$ such that $l$ is in $GB$, the cube view is computed using derived expressions containing aggregations and joins. For instance, a totally materialized data cube with $n$ dimensions, each one having $k_i$ levels, would require, using the summary delta method, computing $\Pi_{i=1}^{n} k_i$ aggregates and $n$ joins with the dimension tables, in case of a DelInstance operation. Using our method, neither aggregates nor joins must be computed. This can be seen in the following example.

Example 13 Figure 8 shows the graph for the plan derived for the data cube of example 9, for a DelInstance(Product, ItemId, $i_2$) update. The nodes inside a box represent the cube views such that $l \in GB$. The sub-lattice indicates which predecessors must be chosen. We can proceed analogously for the case of an AddInstance operation.

5 Discussion

As dimensions represent the framework within which factual data is summarized for analysis, it is our belief that changes in analysis requirements and/or in the structure of the data sources, almost always imply changes in the dimensions of the model. These changes are not limited to the addition or deletion of attributes, but they may also involve the hierarchical structure according to which dimensions are organized. All these kinds of dimension updates, together with the data cube maintenance under them, are poorly supported (or not supported at all) in current commercial systems. Our focus in this paper has been to introduce the problem and present a framework for it.

Several multidimensional models have already been presented [9, 12, 2]. We chose the work by Cabibbo and Torlone [3] as a starting point for our model. We developed a set of operators for structural and instance updates over dimensions. We also presented algorithms to perform maintenance when using ROLAP storage of the data cube, under both structural and instance updates.

Mumick et al [14], proposed the Summary Delta Method for incremental maintenance of a set of materialized aggregate views defined over the same base table. Their approach is applied to generalized cube views, which differ from our cube views in their aggregate functions and the joins they perform with dimension tables. This work focuses mostly on updates to the fact table, but has a brief discussion of instance updates to the dimension tables. In the framework presented there, maintenance of the cube views is performed separately for each derived table, without taking into account the view lattice. We devised instead an algorithm that takes advantage of the view lattice to prevent, whenever possible, joins and aggregate com-
putations.
In future work we will give a more in-depth treatment of the update operators, which we are in the process of implementing. We are also defining higher level operators, such as re-classifying a level within a dimension hierarchy, and maintenance algorithms for them.

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

References