# Supporting Dimension Updates in an OLAP Server

Alejandro A. Vaisman[1], Alberto O. Mendelzon[2], Walter Ruaro[1], and Sergio G. Cymerman[1]

[1] Universidad de Buenos Aires
{avaisman,wruaro,scymer}@dc.uba.ar
[2] University of Toronto
mendel@db.toronto.edu

**Abstract.** Commercial OLAP systems usually treat OLAP dimensions as static entities. In practice, dimension updates are often needed to adapt the warehouse to changing requirements. In earlier work, we defined a taxonomy for these dimension updates and a minimal set of operators to perform them. In this paper we present *TSOLAP*, an OLAP server supporting fully dynamic dimensions. *TSOLAP* conforms to the *OLE DB for OLAP* norm, so it can be used by any client application based on this norm, and can use as backend any conformant relational server. We incorporate dimension update support to *MDX*, Microsoft's language for OLAP, and introduce *TSShow*, a visualization tool for dimensions and data cubes. Finally, we present the results of a real-life case study in the application of *TSOLAP* to a medium-sized medical center.

## 1 Introduction

The term OLAP (On Line Analytical Processing) refers to data analysis over large collections of historical data(*data warehouses*), supporting the decision-making process, allowing the 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.). Several formal models for OLAP applications have been proposed [4,2,1]. Most of these are based on the original idea of the "star schema" [8], in which data is stored in sets of *dimension* 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. In practice, however, the evolution of the information stored in the warehouse often requires updating some dimensions due to changing user requirements or changes in the data sources.

In this paper we present an OLAP server supporting dimension updates and view maintenance under these updates, built following the *OLE DB for OLAP* [10] proposal. We extend *MDX*, Microsoft's language for OLAP with a set of statements supporting dimension update operators. We also introduce a visualization tool for dimensions and data cubes. We present the results obtained in a real-life case study, a medical center in Argentina. These results suggest that

A. Banks Pidduck et al. (Eds.): CAISE 2002, LNCS 2348, pp. 67–82, 2002.

execution times are compatible with real-life application requirements. Moreover, we show that our view maintenance algorithm largely outperforms the Summary-Delta algorithm for maintaining summary tables in a data warehouse [11].

The remainder of the paper is organized as follows: in Section 2 we review the multidimensional model and dimension updates; in Section 3 we describe our implementation in detail. Section 4 presents the case study and discusses the results. We conclude in Section 5.

## 2 Multidimensional Model and Dimension Updates

In previous work, Hurtado *et al.* [6,7] introduced a multidimensional model that includes a framework for dimension updates and a set of dimension update operators. Due to space limitations we will only give a quick overview of that model, and refer the reader to these works for details.

A *dimension schema* is a directed acyclic graph $G(L, \preceq)$ where each node represents a dimension level, and $\preceq$ is a relation between levels such that its transitive and reflexive closure, $\preceq^*$, is a partial order with a unique bottom level $l_{inf}$, and a unique top level *All*.

An *instance* of a dimension is an assignment of a set of *elements* to each dimension level. Moreover, between every pair of levels $l_i$ and $l_j$ such that $l_i \preceq l_j$ there is a function $\rho_{l_i}^{l_j}$ called *rollup*. A dimension instance must satisfy the *consistency condition*, meaning that for every pair of paths from $l_i$ to $l_j$ in $\preceq$, the composition of the rollup functions yields identical functions.

*Example 1.* Figure 1(a) shows dimension schema for a dimension *Geography,*, where for $L = \{city, province, country, All\}$. Relation $\preceq$ contains the following pairs: $city \preceq province, province \preceq country, country \preceq All$. Figure 2(a) shows an instance of the dimension.

Factual data is stored in *fact tables*. Given a set of dimensions $D$, a fact table has a column for each dimension in $D$ and one or more columns for a distinguished type of dimension called $Measure$. A *base fact table* is a fact table such that its attributes are the bottom levels of each one of the dimensions in $D$. A *multidimensional database* is a set of dimensions and fact tables. A *cube view* is a view computed by aggregating data over the measure of a base fact table.

### 2.1 Dimension Updates and View Maintenance

We will now briefly review the set of operators that modify either the schema or an instance of a given dimension. There are two kinds of update operators: structural operators, that modify the schema, and instance operators, that modify an instance.

The *structural* operators are the following. *Generalize* adds a new level above a given one. *Relate* links two independent levels in a dimension. *Unrelate* deletes an edge between two levels. *DelLevel* deletes a level and its rollup functions. *Specialize* adds a new level to a dimension, below the bottom level $l_{inf}$. Although

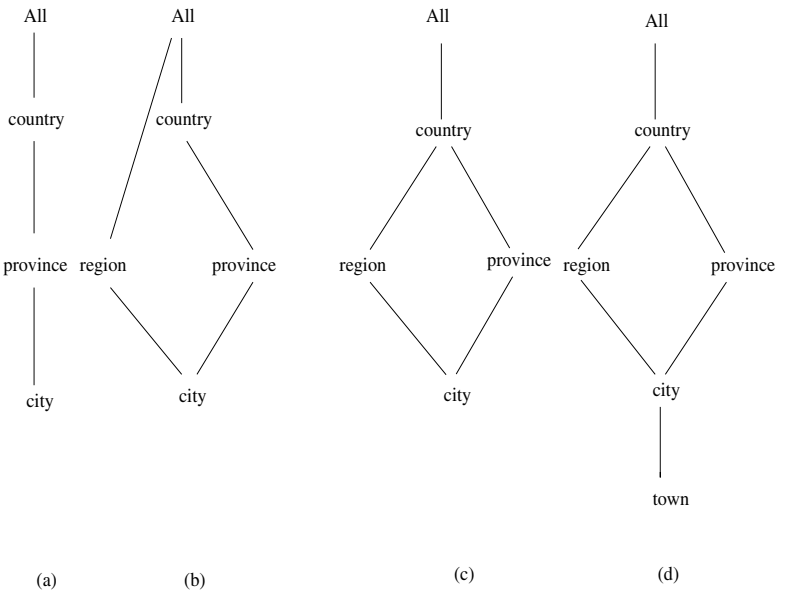


**Fig. 1.** Dimension *Geography*, and a series of updates

*Specialize* can be defined in terms of some of the operators above [13] this would lead to an inefficient implementation so we prefer to treat it as a separate operator. The *instance* operator *AddInstance* inserts a new element into a level, while *DelInstance*, deletes an element from a level.

*Example 2.* Figure 1 shows a sequence of structural dimension updates to the *Geography* dimension. Figure 1(b) depicts a generalization of level *city* to level *region*. Figure 1(c) shows the dimension schema after relating levels *region* and *country* (i.e. this defines that one region cannot belong to two different countries). Finally, Figure 1(d) shows a specialization of level *city* to a finer grain, represented by the level *town*. Deleting *town* would yield the dimension of Figure 1(c). Figure 2(b) depicts the dimension instance after having deleted the city *San Juan City,* and added *Mendoza City,* to the instance of Figure 2(a).

Many common updates to dimensions would result in long sequences of primitive instance updates. Thus, the works cited above define *complex instance update operators*, capturing such common sequences and encapsulating them in a single operation [6,7].

When an update to a dimension occurs, any cube views that have been materialized in the data warehouse must be updated accordingly. Previous works [6,13] present algorithms that outperform traditional incremental maintenance algorithms for materialized views with aggregates [11] by exploiting the special structure of dimension updates. In Section 4 we compare the performance of these algorithms for the $DelInstance$ operation and the $SUM$ aggregate function, against a summary-delta algorithm developed by Mumick et al [11].

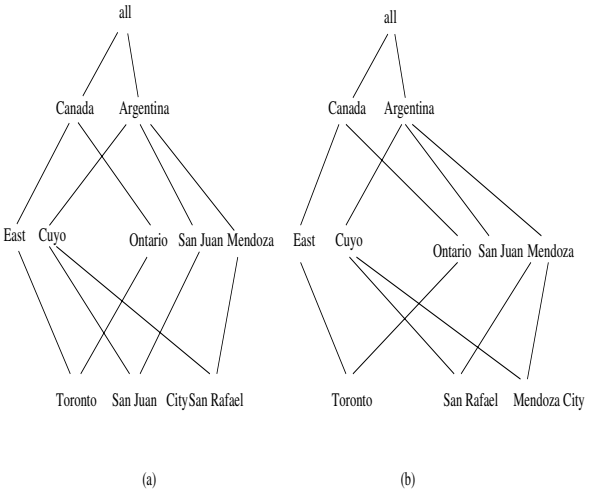


**Fig. 2.** (a) Initial dimension (b) Dimension after adding and deleting instances

### 2.2 Mapping Dimensions to Relations

In previous work [7] Hurtado and some of the authors discussed different ways in which dimensions can be represented in the relational model. The relational representation of a dimension schema that we adopted in the implementation discussed in Section 3 is a relation containing one attribute for each dimension level; each dimension instance contains a tuple for each element to which no other element in the dimension instance rolls up.

Formally, let us denote the relational representation of a dimension $d$ as a pair $R_d = (S_d, T_d)$, where $S_d = (rname, A, \mathcal{F})$ is the schema of the relation; $rname$ is the name of the dimension, and also the name of the relational schema, $A$ is the set of attributes of the relational schema, and $\mathcal{F}$ is a set of functional dependencies such that $dom(\mathcal{F}) \cup ran(\mathcal{F}) \subseteq A$. $T_d$ is the set of tuples in the relation representing $d$. A tuple $R_d = (S_d, T_d)$ represents the dimension, where: (a) $S_d = (rname, A, \mathcal{F})$ is the dimension schema, such that: $rname$ is the name of the dimension; $A$ contains an attribute $l$ for each level $l \in L$; $\mathcal{F}$ contains a functional dependency $l_a \rightarrow l_b$ for each pair of levels $l_a, l_b \in L$ such that $l_a \preceq l_b$; (b) $T_d$, is the set of tuples in the relation. Let us define the leaves of a level $l \in L$, $Leaves(l)$, as the set of elements in a dimension level not reached by any other element below them in the dimension instance. For every level $l$, and for every element $e \in Leaves(l)$, there is a tuple $t$ in $T_d$ such that $t(l_i) = \rho^{*\,l_i}_{\ l}(e)$, if $l \preceq^* l_i$, or $t(l_i) = null$ otherwise, where $\rho^*$ is the transitive and reflexive closure of $\rho$.

We assume that the functional dependencies are only applied over non-null values, i.e., a null in a level $l_a$ can be related to two different elements in $l_b$ even if we have $l_a \rightarrow l_b$ in $\mathcal{F}$.

*Example 3.* The denormalized representation of the dimension depicted in Figure 2(a) is the following:

| city | province | region | country |
|---|---|---|---|
| *Toronto* | *Ontario* | *Cuyo* | *Canada* |
| *SanJuanCity* | *SanJuan* | *Cuyo* | *Argentina* |
| *SanRafael* | *Mendoza* | *Cuyo* | *Argentina* |

## 3 Implementation of Dimension Updates

In this section we present a relational implementation of the model introduced in Section 2. We decided to build an implementation that could be accessed by multiple clients and served by multiple back-ends using reasonably standard interfaces. We chose for this purpose Microsoft's *OLEDB for OLAP* set of interfaces. To express dimension updates, we extended the *MDX* language, the language proposed by Microsoft as a standard for multidimensional database access [10]. The reason for this choice was the wide set of OLAP vendors supporting OLEDB for OLAP, including companies such as Business Objects, SAS Institute, Cognos, and Hyperion, among others. A commitment to this standard was subscribed in the Data Warehouse Alliance [9]. Further, the recently released *XML for Analysis* protocol extends OLE DB for OLAP and OLE DB for DM, allowing standard data access between a client and a provider over the web. However, an implementation could also be built on alternative proposals such as JOLAP [12], a Java-based OLAP API initiative led by ORACLE and IBM.

Roughly speaking, *OLE DB* is a set of low level interfaces for access and manipulation of different data types using the OLE Component Object Model (COM). Thus, OLE DB is more powerful that the well-known ODBC because it is not restricted to relational data. In an OLE DB architecture, a *consumer* is any object consuming an OLE DB interface, while a *provider* is a software component which offers OLE DB interfaces. *OLE DB for OLAP* is an OLE DB extension allowing access and manipulation of multidimensional data like cubes, dimensions, levels and measures, no matter how these data are physically stored. For querying multidimensional data, OLE DB for OLAP employs *multidimensional expressions* written in the *MDX* query language.

Data Cubes are created in MDX using the DDL(Data Definition Language) statement `CREATE CUBE`. However, MDX does not provide update statements for dimensions. Once dimensions and data cubes are created, they remain unchanged "forever." If a situation requiring a dimension update arises, the dimension must be rebuilt from scratch, along with the data cube. We would like to be able to update dimensions and data cubes as soon as it is required, without discarding any current data. Moreover, MDX does not support multiple-path hierarchies (i.e., hierarchies where there are at least two paths between some pair of levels). Instead, we are forced to treat such hierarchies as as a set of single-path hierarchies, which seems rather unnatural. We believe that from the analyst's point

of view, it is important to address multiple hierarchies in a more natural way, providing a higher level of abstraction.

In order to give a solution to these drawbacks, we developed an OLE DB for OLAP data provider called *TSOLAP* that supports dimension updates and performs incremental view maintenance under these updates, together with an MDX extension denoted *MDDLX*.

### 3.1 TSOLAP Architecture

We considered four possible architectures for bringing dimension updates into OLAP: (a) building an application implementing dimension updates over a relational database; (b) modifying an existing OLAP server; (c) developing a hybrid architecture by storing the data cube in a commercial OLAP server, and a catalog in a relational database; (d) implementing a ROLAP server.

We discarded alternative (a) because it meant dimension updates could only be performed by the application program. Alternative (b) was not practical given that we had no access to the internals of an OLAP server. Alternative (c) would not allow maintenance of materialized cube views, as the data cube would be stored in the OLAP server. We chose alternative (d), with the architecture shown in Figure 3.

The ROLAP repository is built on top of a Relational Database. We extended OLE DB for OLAP with a Data Definition Language(MDDLX), and introduced a layer between OLE DB for OLAP and the data source for implementing dimension updates and view maintenance. The data sources can be accessed either through OLE DB or via ODBC (using MSDASQL, an OLE DB provider for ODBC). Configuring the ROLAP server for ODBC allows accessing a wide set of databases. Figure 3 shows that an application could also connect to an OLE DB for OLAP provider via ADO, a set of components and objects allowing data access using a high level programming language. This would require implementing the corresponding ADO interfaces.

### 3.2 Adding Dimension Update Support to MDX

With the support of the architecture described in the previous subsections, we developed a *Multidimensional Data Definition Language* which we called *MDDLX,* an extension to MDX supporting the model of Section 2. We provide statements for the primitive structural and instance update operators, as well as for the complex operators, remaining as faithful as possible to the formal definition of these operators. We also provide a limited `SELECT` statement for displaying the results, although this was not the focus of our work. The `CREATE CUBE` clause creates a data cube. The following statement creates a data cube *Services* for the case study we will present in Section 4.

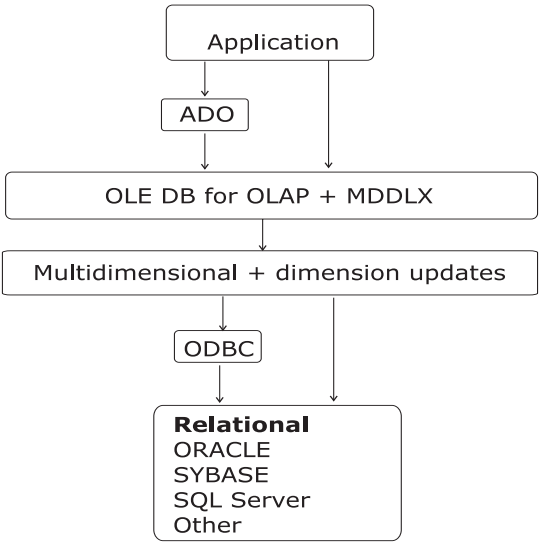


**Fig. 3.** TSOLAP architecture

```
CREATE CUBE Services (
  DIMENSION Doctor BOTTOM LEVEL doctorId TYPE CHAR(6),
  DIMENSION Procedure BOTTOM LEVEL procedureId TYPE CHAR(6),
  DIMENSION Patient BOTTOM LEVEL patientId TYPE CHAR(6),
  TIME DIMENSION Time GRANULARITY DATETIME FROM 01/01/2000 00:00:00
   TO 30/06/2000 23:00:00,
  MEASURE qty TYPE NUMERIC(5,0) FUNCTION SUM,
  MEASURE value TYPE NUMERIC(10,2) FUNCTION SUM)
FROM TABLE data_clinic
WITH MATERIALIZE
```

Here, *data_clinic* is the table from which data is downloaded, with schema (Doctor, Procedure, Patient, Date, qty, value). Attributes corresponding to dimensions must have the same name as the dimension they represent. It is assumed that *data_clinic* has gone through the data extraction and cleaning processes. This table may also include columns that will not be loaded into the cube. The base fact table for the cube will be generated at cube creation time as a materialized view with the least level of aggregation. A new dimension is created for each `DIMENSION` statement, such that a value in the corresponding column of *data_clinic* becomes a value in the bottom level of the dimension. In summary, after the `CREATE CUBE Services` is executed we will have four one-level (plus the distinguished level *All*) dimensions, *Doctor, Procedure, Patient* and *Time*. The `TIME` dimension is generated on-the-fly, taking into account the time column in the table *data_clinic*, which is mandatory. Then, the base fact table will be populated with data facts in the interval defined in the the `FROM`

clause. Further, the `WITH MATERIALIZE` clause specifies that cube materialization is required. Thus, the data cube will contain $2^n$ views, where $n$ is the number of dimensions. Two measures were created, the quantity and the dollar value of the service delivered. Any subsequent dimension update will also require view maintenance. We only support full view materialization at this time. This means that all possible aggregations are created either at cube creation time, or when a dimension update occurs. Moreover, although the syntax allows normalized and denormalized representations, only the latter is currently supported.

The `CREATE CUBE` statement creates dimensions with just two levels, one of them being *All*. To put the dimension in its desired form, once it is created it must be grown incrementally using the dimension update statement `ALTER DIMENSION` (see Section 4). The complete syntax of MDDLX is described in the full paper [14] .

### 3.3 Using TSOLAP

We claimed that any client tool supporting OLEDB for OLAP could be used to display multidimensional data stored in TSOLAP. To show this, we execute our statements using a preexisting client tool, an OLEDB for OLAP consumer called *DataSetViewer*, provided by Microsoft as part of *MDAC2.0* (Microsoft Data Access Components). The *DataSetViewer* allows editing a *MDDLX* query and displaying query results.

An important objective of our work is enhancing the user's capabilities for data analysis. Along these lines, we built a client tool called TSShow for visualizing the structure and instances of the dimensions of every cube in the system. TSShow accesses the catalog tables to display the system's metadata, like cubes, hierarchies, levels and so on. Information about the dimension instances is retrieved from the dimension tables themselves. This tool becomes important in an environment supporting schema and instance updates. Although most of the commercial OLAP systems provide a visualization tool, TSShow not only displays the dimension's structure and instances, but also the rollup functions which hold between elements in the dimension's levels. Figure 4 presents a TSShow screen displaying the cubes and the dimensions in the system. We can see that two cubes were created, *Salescube* and *Services* (see Section 4). The hierarchies of the dimensions are also displayed.

## 4 A Case Study: A Medical Data Warehouse

In Section 3 we presented our implementation of the multidimensional model introduced in Section 2, called TSOLAP. In this section, we apply TSOLAP to a real life case study, a medical center in Argentina. We introduce the problem and describe how the cube and dimensions were built. After this, we discuss different ways in which TSOLAP could be applied to the case study, establish the goals of our experiments, and the hardware we used for the tests. Finally, we present our experimental results.

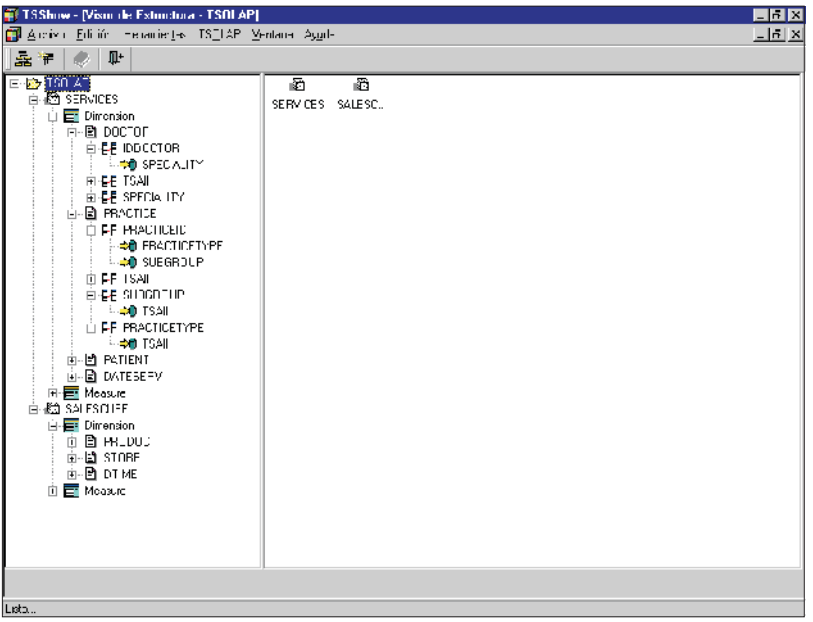


**Fig. 4.** Cube and dimension information with *TSShow*

### 4.1 The Problem

We tested the model and its implementation on a real case, a medical center in Buenos Aires, using six months of data about medical procedures performed on patients. Each patient receives different services, including radiographies, electrocardiograms, and so on. These services are called "Procedures," and are grouped into a classification hierarchy. For instance, a procedure like "Special Radiography" belongs to the subgroup "Radiography" and is further classified into the group denoted "Radiology". Medications given to a patient and disposable supplies are also considered "Procedures." The data were extracted from several tables in the operational database. Taking into account the available data, we designed the data warehouse as follows (see Figures 5 and 6).

A dimension *Procedure,* with bottom level *procedureId,* and levels *procedureType, subgroup* and *group*, gives information about the different procedures available to patients. *Patient*, with bottom level *patientId*, represents information about the person under treatment. As data about the age and gender of the patient are available, we also defined the dimension levels *yearOfBirth* and *gender*. Moreover, we found it interesting to analyze data according to age intervals, represented by a dimension level called *yearRange*. Patients are also grouped according to institutional affiliation. This information could be useful e.g. to categorize patients delivered by various health insurance institutions. Moreover, these institutions are grouped into types such as private companies, unions, and so on. Dimension *Doctor* gives information about the available doctors (identi-

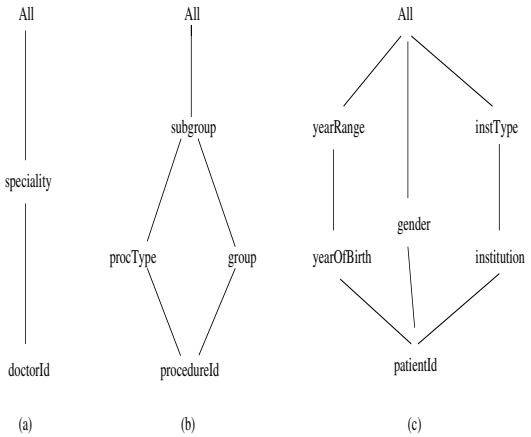


**Fig. 5.** Case study: Dimensions *doctor* and *time*

fied by *doctorId*) and their *specialities* (a level above *doctorId*). Finally, there is a *Time* dimension, as explained in Section *addingdimup*.

When the design was completed, we were ready to create the data cube using MDDLX statements. The `CREATE CUBE` statement in Section 3.2 creates the cube from a table *data_clinic*, with data from the first six months of the year 2000 (631,000 records). Each record in this table contains information about a *procedure* conducted on a certain *patient* by a given *doctor* on a given *date*.

It is now possible to update these dimensions using MDDLX statements, in order to obtain the dimensions depicted in Figures 5 and 6. For example, the following statement generalizes level *patientId* to level *gender*, which is of type `CHAR(1)` ('M' or 'F' ), using the rollup function specified by the table `data3gidintengender`. This table has two columns, one named *patientId* and the other named *gender*, with *patientId* being the key, so each tuple stores the gender of a patient. The complete sequence of statements used to set up the dimensions can be found in the full paper [14].

```
ALTER DIMENSION Services.Patient
GENERALIZE LEVEL patientId
   TO LEVEL gender TYPE CHAR(1)
   USING ROLLUP FUNCTION data3gidintengender
```

### 4.2 What Can We Do with Dimension Updates?

We argue that building dimensions using our approach is, most of the time, more efficient and flexible than building a dimension from scratch every time an update occurs. Moreover, it is possible to add or remove elements from dimensions, or change classification levels in order to query hypothetical database states. Some

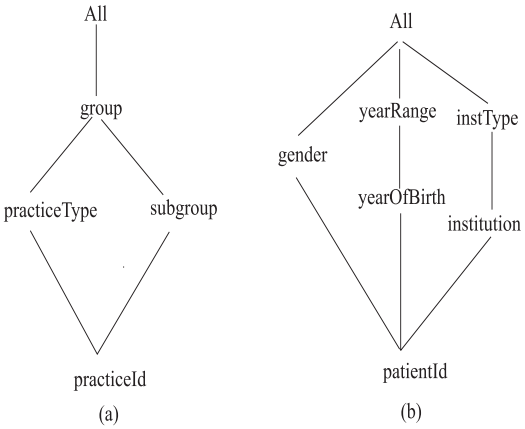


**Fig. 6.** Case study: Dimensions *procedure* and *patient*

examples of these kinds of situations are: (a) in a clinic like the one described in this study, dimension instances are updated all the time. For example, new doctors are hired or leave frequently and new patients are serviced every day. On the other hand, instance updates to the *Procedure* dimension are less frequent; (b) modifying the "yearRange" field in the *Patient* dimension allows finding out which age range is getting more services. This can be done by deleting the *yearRange* level, and then generalizing it again using a different (prepared off-line) rollup function. In a state-of-the-art OLAP system, this would require rebuilding the data cube once for each range test; (c) generalizing the *doctorId* level in dimension *Doctor* to level *doctorAgeRange* is useful to analyze the number of patients served depending on the doctors' age; (d) the model allows inserting, in an on-line fashion, new patients, institutions, institution types, and so on; (e) simulation data could be inserted on-line in order to query different hypothetical database states. Hypothetical situations could be easily modeled by replacing the actual rollup functions with the ones we wish to test. For instance, we could delete the level *yearOfBirth* and generalize level *patientId* again, with data such that seventy percent of the patients are more than sixty years old.

### 4.3 Objectives and Description of the Experiments

From the discussion in Subsection 4.2 it follows that TSOLAP is a very useful tool for data analysis. However, we must show that our approach can reach a performance that can cope with the requirements of everyday applications. Thus, getting a set of data large enough to allow representative results was a requirement. We used six months of data, involving almost 631,000 records; we partitioned this set into the six subsets shown in the table of Figure 7 in order to run the tests over each one of them, to test the performance of the system as the size of the data cube changed. Note that the six subsets contain the same number

| Case # | From | To | # of tuples in FT | # Patients | # Doctors | # Procedures |
|---|---|---|---|---|---|---|
| 1 | 1/1/2000 | 1/31/2000 | 90825 | 6790 | 367 | 3750 |
| 2 | 1/1/2000 | 2/29/2000 | 178698 | 6790 | 367 | 3750 |
| 3 | 1/1/2000 | 3/31/2000 | 270127 | 6790 | 367 | 3750 |
| 4 | 1/1/2000 | 4/30/2000 | 374674 | 6790 | 367 | 3750 |
| 5 | 1/1/2000 | 5/31/2000 | 501628 | 6790 | 367 | 3750 |
| 6 | 1/1/2000 | 6/30/2000 | 630844 | 6790 | 367 | 3750 |

**Fig. 7.** Data sets

of patients, doctors and procedures, because we tested the operators using all the elements in the domains of the rollup functions. In the example above (generalization from *patientId* to *gender* using the rollup table `data3gidintengender`), although the table holds the gender of all the patients, only doctors who actually delivered services before January 31st will be generalized.

In order to test the performance of dimension updates, we executed a set of MDDLX commands over the data cube described in Section 4.1. Our intuition was that performance could be strongly influenced by the order in which operations are performed. Thus, we decided to perform the dimension updates in two different sequences: in the first one, we updated the dimensions in the following order: *Patient*, *Doctor*, *Time* and *Procedure*; in the second sequence, we first performed all the updates over the *Time* dimension, then the ones over *Doctor*, *Procedure* and *Patient*, in that order. Thus, for instance, when performing a generalization over *Procedure*, more materialized views must be updated in the first sequence than in the second one. Both sequences can be found in the full paper.

Our second goal was testing the influence of view maintenance overhead on dimension update performance. To meet this goal, we created the same cube described above, but with the `NO MATERIALIZE` option, and executed the first sequence of updates. Of course, there is no reason for executing both sequences, because no view must be updated in this case.

The third goal was studying query performance when no view materialization is done. To this effect we ran the query " *list the total number of procedures by doctor, subgroup and institution type*" under full materialization, and computing the aggregation on-the-fly. This query involves taking the join of three dimensions of the cube.

Finally, we were interested in comparing the performance of the maintenance algorithm introduced in previous work [6], against a non-optimized algorithm like the standard Summary-Delta method [11]. We performed the tests for a `DELETE INSTANCE` update. We used only three months of data(270,000 tuples in the base fact table), since we expected a non-optimized algorithm on the fully materialized data cube to be too inefficient to run over the full data set. We created two data cubes, one with aggregate function `SUM`, and one with `MAX`. The latter allows no optimization because base data must always be accessed (recall that

`MAX` is not self-maintainable with respect to deletions [3]). Thus, view maintenance techniques cannot avoid the joins. We then applied the following updates: generalize level *datetime* to *date,* generalize *procedureId* to *procedureType,* and generalize *procedureId* to *subgroup,* in this order. The tests were carried out by deleting an element in level *procedureId* over each data cube.

**Hardware.** The tests were run on a PC with an Intel Pentium III 600Mhz processor, with 128 Mb of RAM memory and a 9Gb SCSI Hard Disk. The Database Management System was SQL Server 7.0 database running on top of a Windows NT 4 (Service Pack 5) Operating System, We also ran our tests on an ORACLE 8.04 DBMS, but do not report these results because further experimentation seems neeeded.

### 4.4 Experimental Results

In this section we describe the results of our experiments, following the order in which we stated our objectives in Subsection 4.3.

The creation times for fully materialized data cubes ranges from 60 to 470 seconds, while not materialized data cubes are created in less than 4 seconds for the full set of data (600.000 tuples). Figures 8 and 9 depict generalization time, comparing generalizations of fully materialized data cubes at different aggregation levels, for the two sequences described above. Notice that, even when the generalization from level *yearOfBirth* to level *yearRange* is performed *after* the generalization from *patientId* to level *institution* (sequence 2), the former takes less time to perform, because it affects levels located higher in the dimension's hierarchy. The charts show that the behavior of the operators and view updates is close to linear with respect to the number of tuples in the base fact table. Also notice that in Figure 8 the curve corresponding to the generalization over *Patient* is below the two other ones, while in Figure 9 it is above them, reflecting the influence of the number of updated views. The interested reader can find a more comprehensive description of the results in the full paper.

Updates to instances of dimensions are applied once all the views have been materialized(the same occurs in the case of *DelLevel*). Thus, the sequence of operations in these cases is irrelevant. Figure 10 shows the performance of the *DelInstance* operator.

Our second goal was measuring the time consumed by the operators themselves. Thus, as we explained in Section 4.3, we executed the updates over the cube created with the `NO MATERIALIZE` option. The results we obtained had shown that `GENERALIZE` statements were executed in less than fifteen seconds, for the full data set(*i.e.* 6300.000 tuples).

The results presented above show that execution times are compatible with application requirements. However, the tests over the non-materialized cube demonstrate that almost all the processing time is consumed by the view maintenance operations, suggesting that a partially materialized strategy (i.e. an approach like the one proposed by Harinarayan et al [5]) would be the best option when an evolving scenario like the one proposed here is implemented. As this

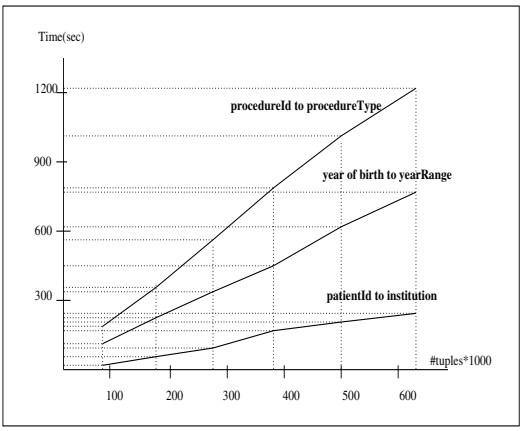


**Fig. 8.** Performance results for *generalize*(sequence 1)

alternative is dependent on query performance, our third experiment focused on studying how an MDDLX query could perform when no view is materialized. We executed the query " *total number of procedures by doctor, subgroup and institution type*" over both cubes under test. For the complete set of data, and no view materialization, the execution time takes two minutes, which seems to be an acceptable result. Of course, queries perform faster under the full materialization strategy, because performing a query under this strategy implies just a sequential scan of the desired view.

Figure 11 gives a summary of the disk space consumed by the database for the six different sets of data, comparing data and index spaces(the data cube contained 630 materialized views after all updates were performed). Notice that the relation between data and index spaces decreases as the data space increases.

Finally, we compared our maintenance algorithm against a non-optimized algorithm for the *DelInstance* operator with different numbers of materialized views. Avoiding unnecessary joins dramatically improves performance, reducing update times by a factor between five and eight.

## 5 Discussion and Summary

We have presented TSOLAP, an implementation of the multidimensional model introduced in previous works by Hurtado and some of the authors [6], and an extension to MDX supporting dimension updates. We also introduced TSShow, a visualization tool for dimensions and data cubes.

We used TSOLAP in a real-life case study. The results showed that our model can be useful not only for database administrators who could avoid rebuilding the multidimensional database each time a dimension is updated, but also for analysts who could benefit from the chance of easily posing hypothetical queries to the system.

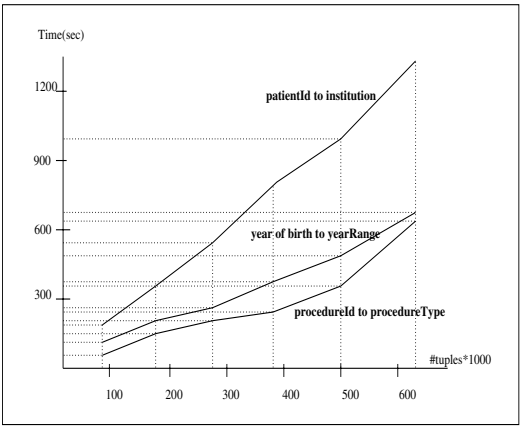


**Fig. 9.** Performance results for *generalize*(sequence 2)

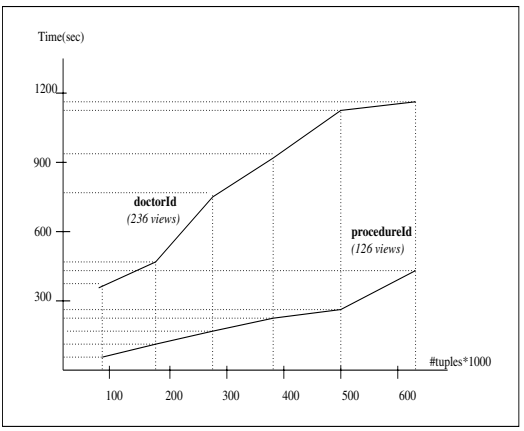


**Fig. 10.** Performance results for *DelInstance*

| Case # | Data space(Mb) | Index space(Mb) | rate |
|---|---|---|---|
| 1 | 324 | 634 | 1.98 |
| 2 | 568 | 1025 | 1.80 |
| 3 | 825 | 1423 | 1.72 |
| 4 | 1108 | 1854 | 1.67 |
| 5 | 1423 | 2333 | 1.63 |
| 6 | 1734 | 2816 | 1.62 |

**Fig. 11.** Data and index disk space

The `DELETE INSTANCE` statement is the most expensive, especially when the aggregate functions of the data cube are `MAX` or `MIN` (aggregate functions which are not self-maintainable). However, most of the execution time was consumed by view maintenance operations, because of our full materialization strategy.

## Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada and the Institute for Robotics and Intelligent Systems.

## References

1. 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. 67
2. A. Golfarelli, D. Maio, and S. Rizzi. Conceptual design of data warehouses from E/R schemes. In *Proceedings of the Hawaii International Conference on System Sciences*, Kona, Hawai, 1998. 67
3. A. Gupta and I. H. Mumick. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, 1999. 79
4. M. Gyssens and L. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of the 22nd VLDB Conference*, pages 106–115, Bombay, India, 1996. 67
5. V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM-SIGMOD Conference*, pages 205 – 216, Montreal, Canada, 1996. 79
6. C. Hurtado, A. O. Mendelzon, and A. Vaisman. Maintaining data cubes under dimension updates. *Proceedings of IEEE/ICDE'99*, 1999. 68, 69, 78, 80
7. C. Hurtado, A. O. Mendelzon, and A. Vaisman. Updating OLAP dimensions. *Proceedings of ACM DOLAP'99*, 1999. 68, 69, 70
8. R. Kimball. *The Data Warehouse Toolkit*. J.Wiley and Sons, Inc, 1996. 67
9. Microsoft Corporation. *Data Warehouse Alliance (DWA2000) (Internet Document http://www.microsoft.com/BUSINESS/bi/dwa/dwa.asp)*, 2000. 71
10. Microsoft Corporation. *OLEDB for OLAP Specification (Internet Document http://www.microsoft.com/data/oledb/olap)*, 2000. 67, 71
11. I. Mumick, D. Quass, and B. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the ACM - SIGMOD Conference*, Tucson, Arizona, 1997. 68, 69, 78
12. Sun Microsystems. *JOLAP: Java OLAP Interface (Internet Document http://jcp.org/jsr/detail/069.jsp)*, 2001. 71
13. A. Vaisman. Updates, view maintenance and time management in multidimensional databases. *Phd Thesis, http://www.cs.toronto.edu/ avaisman/publications*, 2001. 69
14. A. Vaisman, A. O. Mendelzon, W. Ruaro, and S. Cymerman. Supporting dimension updates in a ROLAP server(full paper). In *Internet Document http://www.cs.toronto.edu/ avaisman/publications*, 2002. 74, 76