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 [AGS97,GL96,GMR98,CT98]. Most of these are based on the original idea of the "star schema" [Kim96], 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* [Mic00b] 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

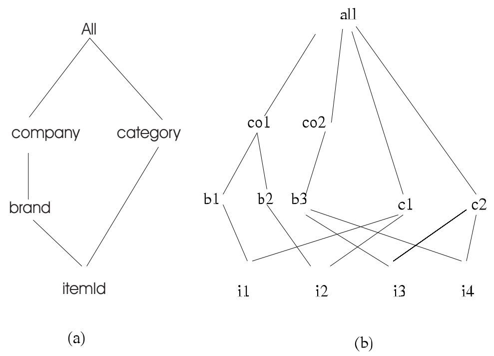

Fig. 1. (a) A dimension schema. (b) A dimension instance.

in a real-life case study, a medical center in Argentina. These results suggest that 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 [MQM97].

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.* [HMV99a,HMV99b] 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.

2.1 Dimensions

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 shows an example of a simple dimension schema and instance for a dimension *Product*. Here, for the dimension schema, $L = \{itemId, brand, company, category, All\}$. Relation $\preceq$ contains the following pairs: $itemId \preceq brand$, $brand \preceq company$, $company \preceq All$, $itemId \preceq category$, $category \preceq All$.

2.2 Fact Tables

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.

Example 2. Consider a set of dimensions *D*= {*Product,Store,Time,Sales*}. The schema and rollup functions for *Product* are given in Figure 1. Schemas for *Time* and *Store* follow from the rollup functions given below . Let us define the following rollup functions: $\rho_{storeId}^{region} = \{s_1 \mapsto r_1, s_2 \mapsto r_2, s_3 \mapsto r_3\}$; $\rho_{day}^{week} = \{d_1 \mapsto w_1, d_2 \mapsto w_1, d_3 \mapsto w_2, d_4 \mapsto w_2\}$ (we have omitted the rollups to *All*). Let us consider the following base fact table, call it *DailySales*:

itemId	storeId	day	sales
i_1	s_1	d_1	10
i_2	s_1	d_1	20
i_2	s_2	d_1	20
i_2	s_2	d_2	40
i_3	s_3	d_3	30

Assuming that the *measure* dimension is *Sales* (with measure level *sales*), the following table is a cube view aggregating data over *itemId, storeId,* and *week:*

itemId	storeId	*week*	sales
i_1	s_1	w_1	10
i_2	s_1	w_1	20
i_2	s_2	w_1	60
i_3	s_3	w_2	30

2.3 Dimension Updates

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 *Specialize* can be defined in terms of some of the operators above [Vai01] this would lead to an inefficient implementation so we prefer to treat it as a separate operator.

Example 3. Figure 2 shows a sequence of dimension updates to a *Geography* dimension. Figure 2(b) depicts a generalization of level *city* to level *region*. Figure

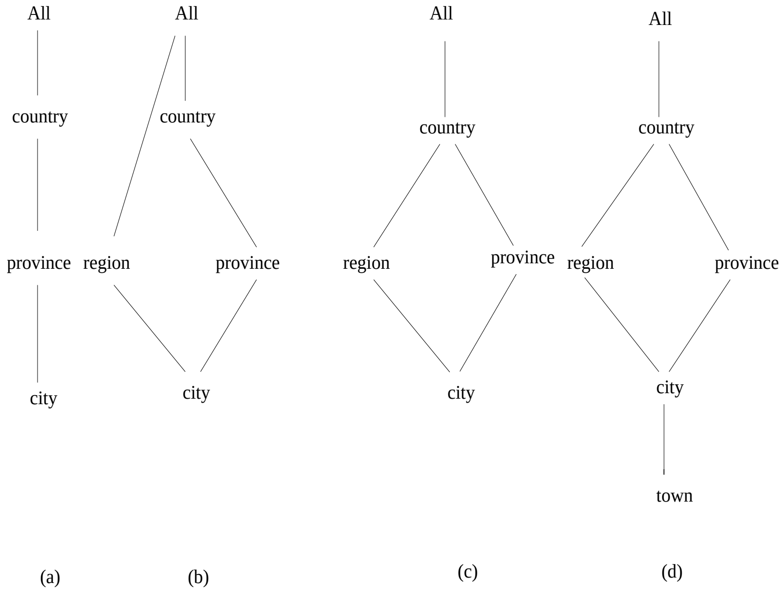

Fig. 2. A series of updates to dimension *Geography.*

2(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 2(d) shows an specialization of level *city* to a finer grain, represented by the level *town.* Deleting *town* would yield the dimension of Figure 2(c).

The instance operators are *AddInstance*, which inserts a new element into a level, and *DelInstance*, which deletes an element from a level.

Example 4. Figure 3(a) shows an instance of the dimension *Geography* above. Figure 3(b) depicts the dimension instance after having deleted the city *San Juan City,* and added *Mendoza City.*

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 [HMV99a,HMV99b].

2.4 View Maintenance

When an update to a dimension occurs, any cube views that have been materialized in the data warehouse must be updated accordingly. Previous works [HMV99a,Vai01] present algorithms that outperform traditional incremental maintenance algorithms for materialized views with aggregates [Qua96,MQM97] 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*

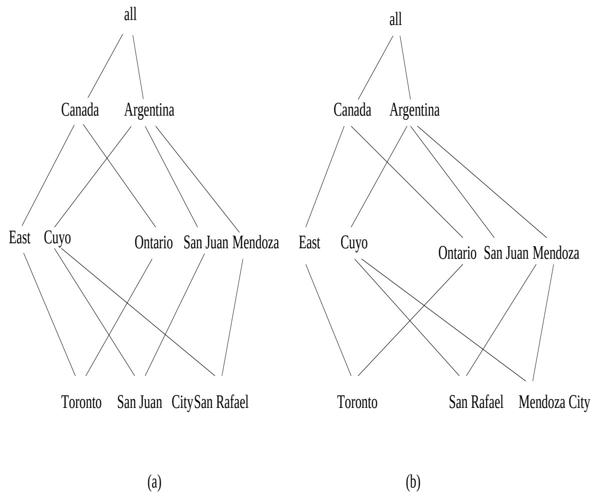

Fig. 3. (a) Initial dimension (b) Dimension after adding and deleting instances.

aggregate function, against a summary-delta algorithm developed by Mumick et al [MQM97].

2.5 Mapping Dimensions to Relations

In previous work [HMV99b] Hurtado and some of the authors discussed different ways in which dimensions can be represented in the relational model. We studied two possible implementations: dimensions stored as a single table (denormalized representation), and dimensions stored as a set of normalized tables (normalized representation). The results showed that the denormalized representation is, in most cases, more efficient than the normalized one. Thus, we adopted it for the implementation we present in Section 3. The denormalized relational representation of a dimension schema 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 dimension is mapped to a denormalized relational representation $R_d = (S_d, T_d)$ in the following way:

(a) The schema $S_d = (rname, A, \mathcal{F})$ of the relation is defined as follows: $rname$ 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) The set of tuples T_d in the relation is defined as follows: 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 defined as follows:

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

The definition above implies that the number of tuples in the relation is given by

$$\sum_l card(Leaves(l));$$

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 $\mathcal{F}$. This allows a relation with nulls in attributes associated to levels below leaf elements, situation that arises when there exists a level $l \neq l_{inf}$ such that $card(Leaves(l)) \geq 2$.

Example 5. The denormalized representation of the dimension depicted in Figure 1(b) is the following:

itemId	brand	company	category
i_1	b_1	co_1	c_1
i_2	b_2	co_1	c_1
i_3	b_3	co_2	c_2
i_4	b_3	co_2	c_2

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 [Mic00b,Mic00c]. 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 [Mic00a]. Further, the recently released *XML for Analysis* protocol extends OLE DB for OLAP and OLE DB for DM, standardizing data access between a client and a provider over the web. However, an implementation could also be built on alternative proposals such as JOLAP [Sun01], a Java-based OLAP API initiative led by ORACLE and IBM.

3.1 Overview of OLE DB for OLAP and MDX

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.

Consider for instance, the data warehouse of Example 2, and the following query over a cube based on the fact table in that example.

```
SELECT {Product.brand.MEMBERS} ON COLUMNS,
           {Time.day.MEMBERS} ON ROWS
     FROM [DailySales]
     WHERE ([Measures].[sales])
```

The query returns the table:

	b_1	b_2	b_3
d_1	10	40	0
d_2	0	40	0
d_3	0	0	30

The data cube specified in the `FROM` clause can be created in MDX using the DDL(Data Definition Language) statement `CREATE CUBE` depicted in Figure 4.

MDX does not provide update statements for dimensions. Once dimensions and data cubes are created, they remain unchanged "forever." This is reflected in the `CREATE CUBE` statement of Figure 4, where all the dimension levels must be defined at cube creation time. 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. For example, in the statement of Figure 4, we can see that in order to build a dimension like the one depicted in Figure 1, we have to define two different hierarchies, denoted *ProdBrand* and *ProdCat* respectively: $itemId \rightarrow brand \rightarrow company \rightarrow All$, and $itemId \rightarrow category \rightarrow All$. 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.

```
CREATE CUBE DailySales (
DIMENSION TIME TYPE TIME,
  HIERARCHY [Weekly],
   LEVEL [all] TYPE ALL,
   LEVEL [day] TYPE DAY ,
   LEVEL [week] TYPE WEEK,
DIMENSION PRODUCT,
  HIERARCHY [ProdCat],
   LEVEL [ProdCat all] TYPE ALL,
   LEVEL [ProdCat category],
   LEVEL [ProdCat itemId] TYPE YEAR,
  HIERARCHY [ProdBrand],
   LEVEL [ProdBrand all] TYPE ALL,
   LEVEL [ProdBrand company],
   LEVEL [ProdBrand brand],
   LEVEL [ProdBrand itemId],
DIMENSION STORE,
   LEVEL [all] TYPE ALL,
   LEVEL region,
   LEVEL storeId,
MEASURE [sales] FUNCTION SUM )
```

Fig. 4. `CREATE CUBE` statement in MDX

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*.

3.2 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(see Figure 5); (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 6.

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

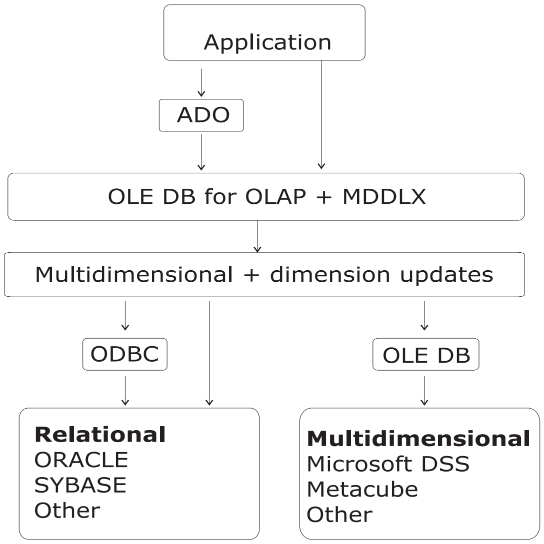

Fig. 5. Hybrid Architecture

provider for ODBC). Configuring the ROLAP server for ODBC allows accessing a wide set of databases.

Finally, Figures 5 and 6 show that an application could 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.

The system manages two kinds of tables: *catalog* tables and *data cube* tables. The latter are created when a data cube is created, and they are eliminated as soon as the data cube they are associated to is dropped. The catalog is stored in the same relational database as the data cube, and is accessed by the TSOLAP routines when an update or query is submitted (see Figure 8 and Section 3.3). For example, tables TSLevels$, TSLevelsd$ and TSHierarchies$, store the dimension levels, the dimension schema, and the dimension hierarchies, respectively. The latter table is needed in order to allow supporting multiple-path hierarchies. Each hierarchy in a dimension is stored as a path between the bottom level and the distinguished level *All.*

The basic functionality needed to comply with the OLEDB for OLAP specifications was implemented in a library named TSOLAP.dll. This library implements the methods which detect a data source, create a new session, and create a new command, among other functions. Library TSCube.dll contains the classes which manage every aspect of the data cube. Every MDDLX command is applied to an object of the class CTSCube, which demands tasks from other objects in the system. CTSRepresentation implements the update operators, CTSDag manages the dimension schema (*i.e.* adds and deletes edges, checks the operator's preconditions, etc.). CTSLex and CTSYacc implement the

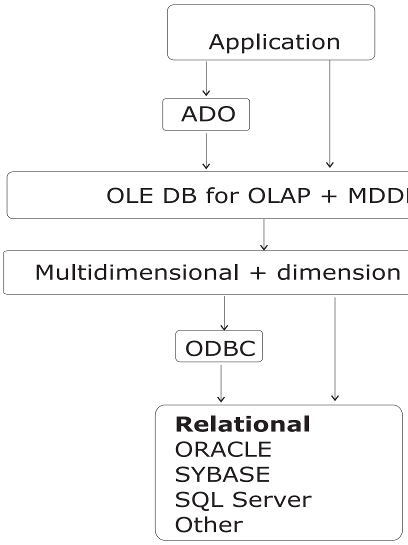

Fig. 6. TSOLAP Architecture

MDDLX parser(using PCLex and PCYacc), and CTSData.dll manages the creation and maintenance of materialized views. In Figure 8 we show how all these libraries interact.

3.3 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 *DailySales*(compare with the MDX statement at the end of Section 2).

```
CREATE CUBE DailySales (
  DIMENSION Product BOTTOM LEVEL itemId TYPE CHAR(6),
  DIMENSION Store BOTTOM LEVEL storeId TYPE CHAR(6),
  TIME DIMENSION Time GRANULARITY DATETIME FROM 01/01/1999 00:00:00
TO 12/31/2000 23:59:59,
  MEASURE qty TYPE NUMERIC(5,0) FUNCTION SUM,
  MEASURE amount TYPE NUMERIC(10,2) FUNCTION SUM)
FROM TABLE data_sales
WITH MATERIALIZE
```

In the statement above, *data_sales* is the table from which data is downloaded, with schema (Product, Store, date, quantity, amount). Attributes corresponding to dimensions must have the same name as the dimension they represent. It is assumed that *data_sales* 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_sales* becomes a value in the bottom level of the dimension. In summary, after the `CREATE CUBE DailySales` is executed we will have three dimensions each one with just one level (plus the distinguished level *All*). The other levels are created with the update statements described below. [1]

The `TIME` dimension is generated on-the-fly, taking into account the time column in the table *data_sales*, which is mandatory. Then, the base fact table will be populated with data facts in the interval defined by the clause `FROM 01/01/1999 00:00:00 TO 12/31/2000 23:59:59.`

The `WITH MATERIALIZE` clause specifies if cube materialization is required. This implies that 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 `CUBE CREATION` 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 statements described below. For instance, the dimension *Product* in Figure 1 could have been built as follows: after creating a dimension with levels *itemId* and *All,* level *itemId* could have been generalized to levels *brand* and *category*, in this order. After this, *brand* could have been generalized to *company.* The statement for the first generalization is :

```
ALTER DIMENSION DailySales.Product
    GENERALIZE LEVEL itemId
    TO LEVEL brand TYPE char(10)
    USING ROLLUP FUNCTION generalizeitembrand
```

The table `generalizeitembrand` has the following tuples: $\{(i_1, b_1), (i_2, b_2), (i_3, b_3), (i_4, b_3)\}$. The complete syntax of MDDLX is described in Appendix B.

3.4 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

[1] At this time, level attributes are not supported in our current implementation.

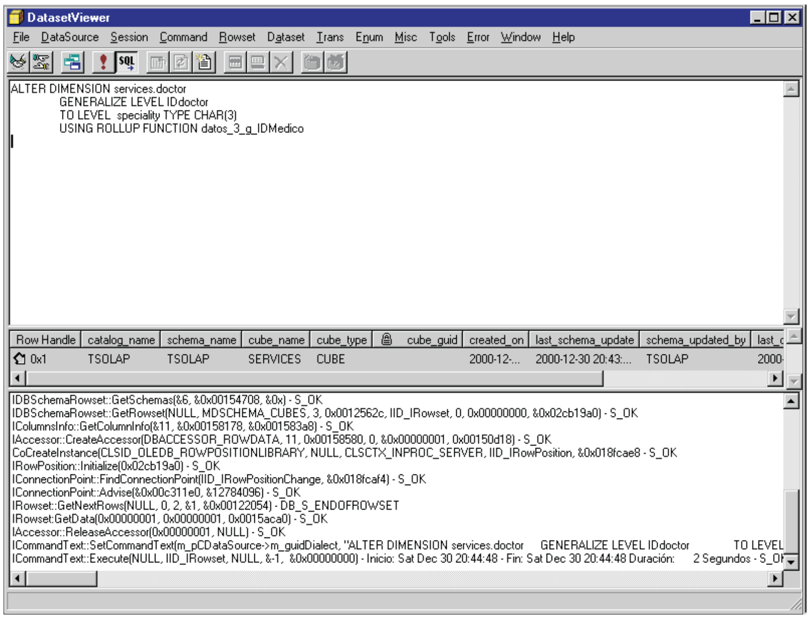

Fig. 7. A `GENERALIZE` command

called *DataSetViewer*, provided by Microsoft as part of *MDAC2.0* (Microsoft Data Access Components). Figure 7 shows a `GENERALIZE` command. The screen is split in three sections: the upper is for editing a *MDDLX* query. The middle one shows query results, either from queries or metadata. The lowest part of the screen shows the query log, query execution time and error codes. Note the log window, showing the sequence of methods of TSOLAP being invoked by the consumer (in this case, *DataSetViewer*) .

Figure 8 shows the sequence of calls which are triggered when a `CREATE CUBE` command is posed to the system (the other MDDLX statements behave in a similar way). Note that after the command is created and the methods *SetCommandText* and *Execute* called, the provider invokes the data cube manager, which assigns responsibilities to the objects in the way described above.

3.5 Visualization:TSShow

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 to visualize of 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

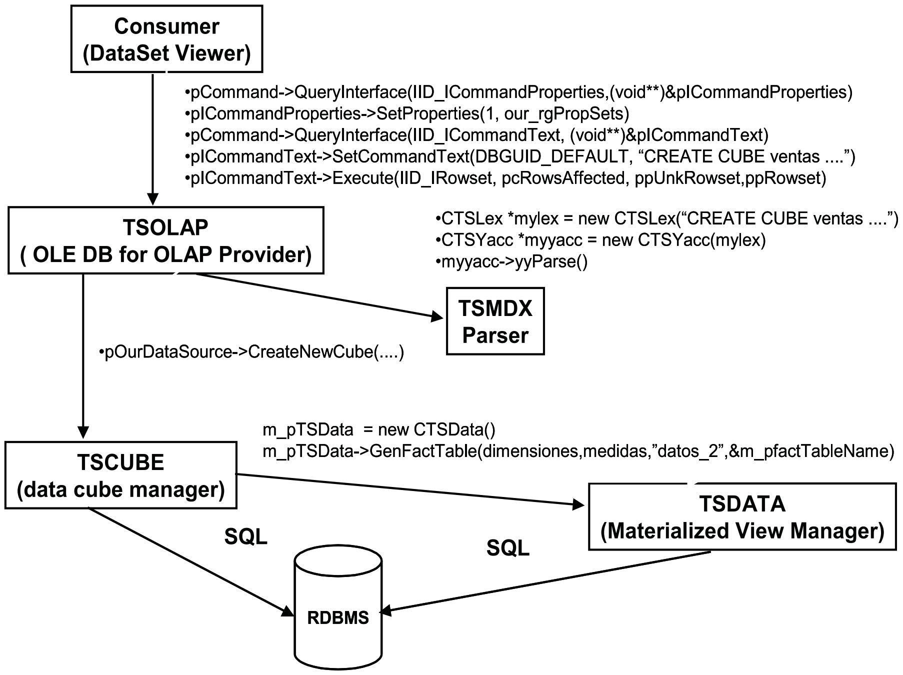

Fig. 8. Execution steps

the dimension's structure and instances, but also the rollup functions which hold between elements in the dimension's levels. Figure 9 presents a TSShow screen displaying the cubes and the dimensions in the system. We can see that two cubes were created, *Salescube* and *Services*. 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.

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

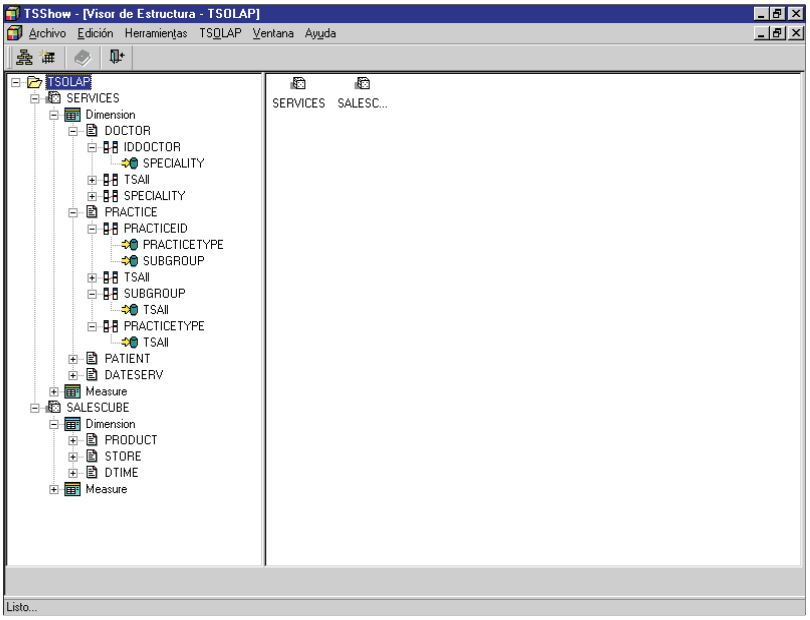

Fig. 9. Cube and Dimension information with *TSShow*

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" is classified as follows :

```
10. Radiology
10.01. Radiography
10.01.01 Special Radiography
```

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 10 and 11).

A dimension *Procedure,* with bottom level *procedureId,* and levels *procedureType, subgroup* and *group*, gives information about the different procedures available to patients.

A dimension *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 (identified by *doctorId*) and their *specialities* (a level above *doctorId*).

The last dimension we designed was *Time*. Recall that the actual values of this dimension are created on-the-fly.

When the design was completed, we were ready to create the data cube using MDDLX statements. The following statement 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*.

```
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
```

This statement generates four one-level(plus "All") dimensions, *Doctor, Procedure, Patient* and *Time*, and a fully materialized data cube (specified by the `WITH MATERIALIZE` option). The cube created contains 2^n views (with $n = 3$, the number of dimensions) with attributes *(procedureId, patientId, All), (All, patientId, doctorId), (procedureId, All, doctorId),* and so on. Two measures were created, the quantity and the dollar value of the service delivered.

It is now possible to update these dimensions using MDDLX statements, in order to obtain the dimensions depicted in Figures 10 and 11. 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.

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

The following `RELATE` statement creates a connection between levels *practiceType* and *group*. Notice the only information needed is the name of the levels

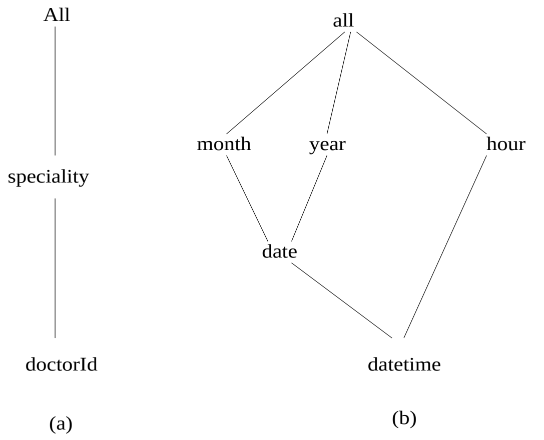

Fig. 10. Case study: Dimensions *Doctor* and *Time*

being related, the name of the cube, and the name of the dimension.

```
ALTER DIMENSION Services.Procedure
 RELATE LEVEL practiceType
   TO LEVEL group
```

The complete sequence of statements used to set up the dimensions can be found in Appendix A.

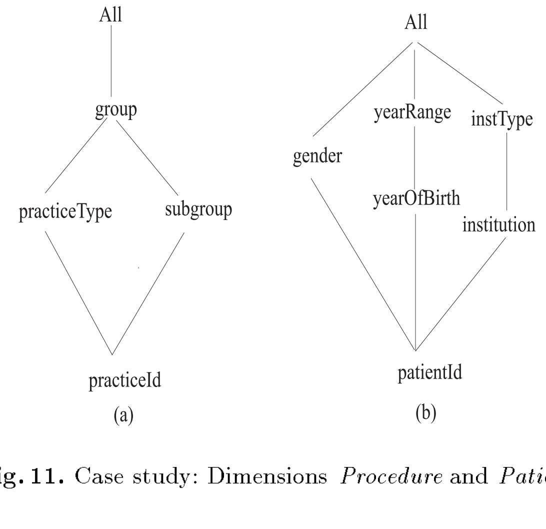

Fig. 11. Case study: Dimensions *Procedure* and *Patient*

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 examples of these kinds of situations are:

- 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.
- 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.
- Generalizing the *doctorId* level in dimension *Doctor* to level *doctorAgeRange* is useful to analyze the number of patients served depending on the doctors' age.
- The model allows inserting, in an on-line fashion, new patients, institutions, institution types, and so on.
- 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 explained in Subsection 4.1 that we used six month of data, involving almost 631,000 records. We partitioned this set into the six subsets shown in the table of Figure 12 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 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 *patienId* 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

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. 12. Data Sets

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 sequence we updated the dimensions in the following order: *Patient*, *Doctor*, *Time* and *Procedure* (notice, however, that some updates over *Patient* occur after updates over the *Time* dimension, see Appendix A for details);
– 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 Appendix A.

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. In MDDLX:

```
SELECT Patient.instType, Doctor.idDoctor, Procedure.subgroup
ON COLUMN
      FROM Services
      WHERE ( qty )
```

As we already explained, TSOLAP materializes every possible view in the data cube, producing 630 materialized views in our case study after all updates are applied. Further, every view is indexed on all of its columns. We wanted to observe the resulting data cube sizes, and how the size of the set of indices relates to the size of the data.

Finally, we were interested in comparing the performance of the maintenance algorithm introduced in previous work [HMV99a], against a non-optimized algorithm like the standard Summary-Delta method [MQM97]. 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 [GM99,MQM97]). Thus, view maintenance techniques cannot avoid the joins. We then applied the following updates: generalize level *datetime* to level *date,* generalize level *procedureId* to *procedureType,* and generalize level *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 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 tables and graphics in Figures 13 to 20 give an idea of the performance of the update operators vs. the number of tuples in the base fact table. Times are expressed in seconds. Figure 13 shows the cube creation time with total view materialization and no materialization at all. Figures 14 to 17 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 15 the curve corresponding to the generalization over *Patient* is below the two other ones, while in Figure 16 it is above them, reflecting the influence of the number of updated views. Moreover, in Figure 14, both curves(corresponding to updates over *Practice*) are shifted upwards with respect to Figure 17, because the number of materialized views is larger at generalization time.

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 20 shows the performance of the *DelInstance* operator. The number of views including the involved attribute is also shown.

Our second goal was measuring the time insumed 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 expression "Time(NM)" in the heading of the table, denotes the results obtained in this way. Results for some `GENERALIZE` statements are depicted in Figure 21.

The results presented above show that execution times are compatible with application requirements. Notice that the higher in the hierarchy the update statements are applied, the faster they perform. 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 [HRU96]) would be the best option when an evolving scenario like the one proposed here is implemented. As this 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. The results are shown in Figure 22. 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 23 gives a summary of the disk space consumed by the database for the six different sets of data, comparing data and index spaces. 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. Figure 24 shows the results. We can see that avoiding unnecessary joins dramatically improves performance. We do not show results for *irrelevant updates* [HMV99a,Vai01]. For these updates, response time is negligible.

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 [HMV99a], 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.

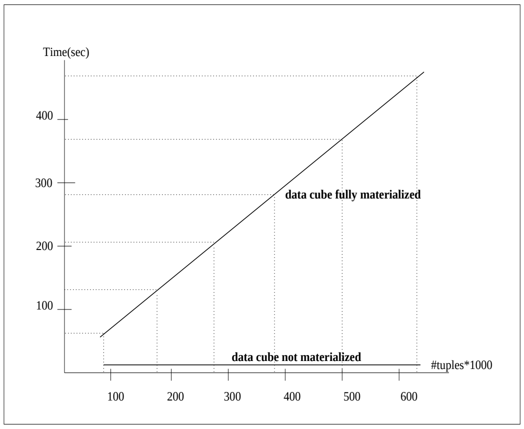

Fig. 13. Data cube creation time

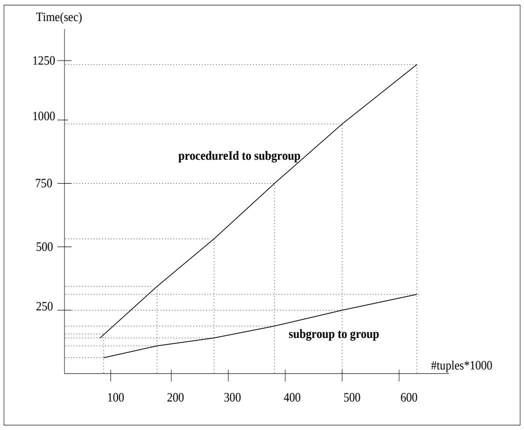

Fig. 14. Performance results for *Generalize*(sequence 1)

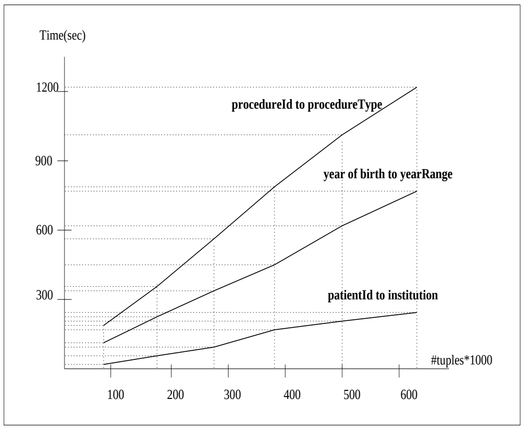

Fig. 15. Performance results for *Generalize*(sequence 1)

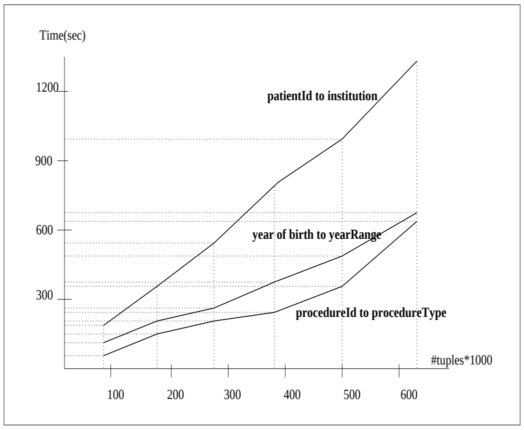

Fig. 16. Performance results for *Generalize*(sequence 2)

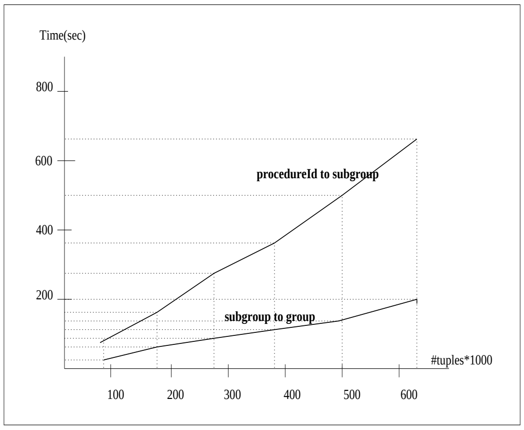

Fig. 17. Performance results for *Generalize*(sequence 2)

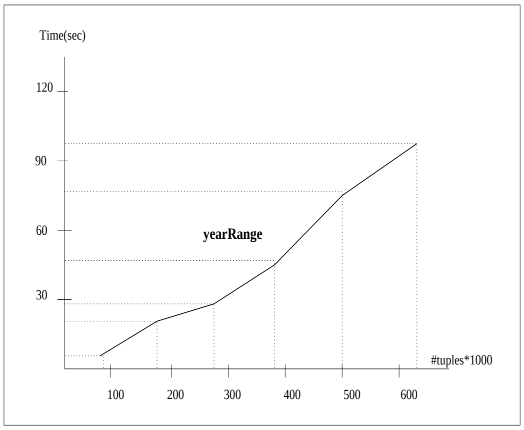

Fig. 18. Performance results for *DelLevel*

Dimension	*From* level	*To* level	# tuples	Time	Time(NM)
Procedure	procedureType	group	90000	25	25
Procedure	procedureType	group	220000	25	25
Procedure	procedureType	group	370000	30	30
Procedure	procedureType	group	600000	35	35

Fig. 19. Performance results for *Relate* (sequence 1)

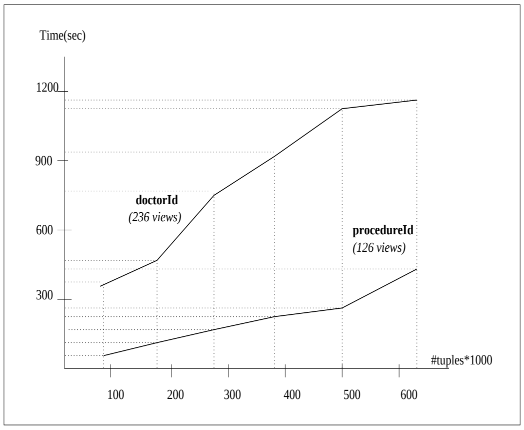

Fig. 20. Performance results for *DelInstance*

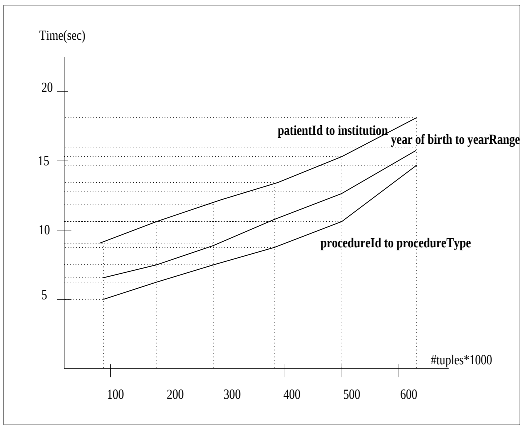

Fig. 21. Performance results for *Generalize* with no view materialization

# of tuples in source	Time(NM)	Time	#tuples in result
90000	$\simeq$0	16	353
160000	$\simeq$0	26	423
230000	$\simeq$0	40	497
370000	$\simeq$0	68	538
600000	$\simeq$0	120	621

Fig. 22. Full vs No Materialization

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. 23. Data and Index disk space

# of Materialized Views	Optimized(sec)	Non-Optimized(sec)
16	14	103
24	29	196
36	75	291
48	99	482

Fig. 24. *Del Instance* Optimized vs. non-optimized

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.

References

[AGS97] R Agrawal, A Gupta, and S. Sarawagi. Modeling multidimensional databases. *IBM Almaden Research Report*, 1997.

[CT98] 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.

[GL96] M. Gyssens and L. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of the 22nd VLDB Conference*, pages 106–115, Bombay, India, 1996.

[GM99] A. Gupta and I. H. Mumick. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, 1999.

[GMR98] 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.

[HMV99a] C. Hurtado, A.O. Mendelzon, and A. Vaisman. Maintaining data cubes under dimension updates. *Proceedings of IEEE/ICDE'99*, 1999.

[HMV99b] C. Hurtado, A.O. Mendelzon, and A. Vaisman. Updating OLAP dimensions. *Proceedings of ACM DOLAP'99*, 1999.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM-SIGMOD Conference*, pages 205 – 216, Montreal, Canada, 1996.

[Kim96] R. Kimball. *The Data Warehouse Toolkit*. J.Wiley and Sons, Inc, 1996.

[Mic00a] Microsoft Corporation. *Data Warehouse Alliance (DWA2000) (Internet Document http://www.microsoft.com/BUSINESS/bi/dwa/dwa.asp)*, 2000.
[Mic00b] Microsoft Corporation. *OLEDB for OLAP Specification (Internet Document http://www.microsoft.com/data/oledb/olap)*, 2000.
[Mic00c] Microsoft Corporation. *OLEDB Programmer's Reference (Internet Document http://msdn.microsoft.com/library/psdk/dasdk/olpr0fqo.htm)*, 2000.
[MQM97] 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.
[Qua96] D. Quass. Maintenance expressions for views with aggregations. In *ACM Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, 1996.
[Sun01] Sun Microsystems. *JOLAP: Java OLAP Interface (Internet Document http://jcp.org/jsr/detail/069.jsp)*, 2001.
[Vai01] A. Vaisman. Updates, view maintenance and time management in multidimensional databases. *Phd Thesis, http://www.cs.toronto.edu/ avaisman/publications*, 2001.

A MDDLX Operators for the Clinic Case Study

In this appendix we give the complete testing sequences used in the case study presented in Section 4. The `DROP CUBE` and `CREATE CUBE` statements are common for both sequences.

```
DROP CUBE Services

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
30/05/2000 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
```

A.1 Testing Sequence 1.

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

2.ALTER DIMENSION Services.Patient
   GENERALIZE LEVEL patientId
      TO LEVEL yearOfBirth TYPE CHAR(3)
      USING ROLLUP FUNCTION data3gidpatientage
```

```
3.ALTER DIMENSION Services.Patient
   GENERALIZE LEVEL patientId
      TO LEVEL Institution TYPE CHAR(4)
      USING ROLLUP FUNCTION data3ginstitution

4.ALTER TIME DIMENSION Services.dayTimeDim
    GENERALIZE GRANULARITY DATETIME TO DATE

5.ALTER DIMENSION Services.Doctor
   GENERALIZE LEVEL IDDoctor
      TO LEVEL speciality TYPE CHAR(3)
      USING ROLLUP FUNCTION data3gIDDoctor

6.ALTER DIMENSION Services.Procedure
   GENERALIZE LEVEL procedureId
      TO LEVEL practiceType TYPE CHAR(5)
      USING ROLLUP FUNCTION data3gprocedureIdtipopre

7.ALTER DIMENSION Services.Procedure
   GENERALIZE LEVEL procedureId
      TO LEVEL subgroup TYPE CHAR(5)
      USING ROLLUP FUNCTION data3gprocedureIdsubgroup

8.ALTER DIMENSION Services.Procedure
   GENERALIZE LEVEL subgroup
      TO LEVEL group TYPE CHAR(3)
      USING ROLLUP FUNCTION data3gsubgroup

9.ALTER TIME DIMENSION Services.dayTimeDim
   GENERALIZE GRANULARITY DATETIME TO HOUR

10.ALTER TIME DIMENSION Services.dayTimeDim
    GENERALIZE GRANULARITY DATE TO YEAR

11.ALTER TIME DIMENSION Services.dayTimeDim
    GENERALIZE GRANULARITY DATE TO MONTH

12.ALTER DIMENSION Services.Procedure
   RELATE LEVEL practiceType
      TO LEVEL group

13.ALTER DIMENSION Services.Patient
   GENERALIZE LEVEL institution
      TO LEVEL instType TYPE CHAR(10)
      USING ROLLUP FUNCTION data3ginstitype

14.ALTER DIMENSION Services.Patient
   GENERALIZE LEVEL yearOfBirth
      TO LEVEL yearRange TYPE CHAR(5)
```

```
      USING ROLLUP FUNCTION data3agerange

15.ALTER DIMENSION Services.practice
     ADD INSTANCE M999999
      INTO LEVEL procedureId
      TO LEVELS ( SubGroup,practiceType )
      VALUES ( M0002, Z0002 )

16.ALTER DIMENSION Services.Doctor
     DELETE INSTANCE 1450
      FROM LEVEL idDoctor

17.ALTER DIMENSION Services.Patient
     DELETE LEVEL yearRange
```

A.2 Testing Sequence 2.

The sequence of updates 1 to 17 described above, is executed in the following order:
`4, 9, 11, 5, 6, 7, 8, 12, 10, 1, 2, 3, 13, 14, 15, 16, 17.`

B MDDLX Syntax

In this appendix we present the complete syntax of the MDDLX language.

- **SELECT statement**
  ```
  SELECT axis_specification ON COLUMNS
  FROM cube_name
  WHERE slicer_specification
  ```
- **Creating Data Cubes**
  ```
  CREATE CUBE cube_name (
  DIMENSION dimension_name BOTTOM LEVEL level_name TYPE data_type
  [REPRESENTATION TYPE [ NORMALIZED | DENORMALIZED ] ]
  [,DIMENSION dimension_name BOTTOM LEVEL level_name TYPE data_type
  [REPRESENTATION TYPE [ NORMALIZED | DENORMALIZED ] ,... ]
  [ [,TIME DIMENSION dimension_name TYPE time_type FROM time_value TO time_value]
  , ...]
  MEASURE measure_name TYPE data_type FUNCTION measure_type
  [,MEASURE measure_name TYPE data_type FUNCTION measure_type , ...] )
  FROM TABLE table_name
  [WITH MATERIALIZE]
  ```
- **Dropping Data Cubes**
  ```
  DROP CUBE cube_name
  ```
- **Dimension creation**
  ```
  ADD DIMENSION cube_name.dimension_name
  BOTTOM LEVEL level_name TYPE data_type [REPRESENTATION TYPE
  [ NORMALIZED | DENORMALIZED ] ]
  FROM TABLE table_name
  ```
  Table_name is a one-column table such that the column name is the dimension's name, and each tuple is an element in the instance set of the bottom level of the dimension.

– *TIME* **Dimension creation**

```
ADD TIME DIMENSION cube_name.dimension_name
GRANULARITY time_type FROM time_value TO time_value
[REPRESENTATION TYPE [ NORMALIZED | DENORMALIZED ] ]
```

The term `time_type` can be one of: `DATETIME, DATE, DAY, WEEK, MONTH, YEAR,` or `HOUR`. This statement automatically generates a `TIME` dimension, in the specified granularity. We can have different `TIME` dimensions, with different granularities and different names. This dimension, again, has only two levels: the bottom level and *All*. For instance, if the user specifies a `time_type` of `DAY` and two dates, say January 1st, 2001 and February 31st, 2001, a time dimension with one level, day, is created, and populated with thirty-one tuples, each for a different day. Of course this does not prevent the user from creating a user-defined time dimension. For example, a user may want to have a level like "quarter" in the time dimension, which the system does not provide.

– **Updating Dimensions**

The `CUBE CREATION` statement creates dimensions with two levels, one of them being *ALL*. A dimension must be built using dimension updates statements.

- ```
  ALTER DIMENSION cube_name.dimension_name
      GENERALIZE LEVEL level_name
      TO LEVEL new_level_name TYPE data_type
      USING ROLLUP FUNCTION table_name
  ```
  Here, table_name is a two-column table representing the rollup function to be applied. The first column holds the instance set of the level to be generalized, while the second column holds the values of the new level.
- ```
  ALTER TIME DIMENSION cube_name.dimension_name
  GENERALIZE GRANULARITY time_type TO time_type
  ```
  The values for the *TIME* dimension are generated on-the-fly, according to the time granularities.
- ```
  ALTER DIMENSION cube_name.dimension_name
  SPECIALIZE TO LEVEL new_bottom_level_name TYPE data_type
  [USING ROLLUP FUNCTION table_name]
  ```
- ```
  ALTER DIMENSION cube_name.dimension_name
      DELETE LEVEL level_name
  ```
- ```
  ALTER DIMENSION cube_name.dimension_name
      RELATE LEVEL level_source_name TO LEVEL level_target_name
  ```
- ```
  ALTER DIMENSION cube_name.dimension_name
      UNRELATE LEVEL level_source_name FROM LEVEL level_target_name
  ```
- ```
  ALTER DIMENSION cube_name.dimension_name
      ADD INSTANCE new_instance_value INTO LEVEL level_name
      [TO LEVELS (level_target1 [, ..., level_targetn ])
      VALUES (instance_target1 [, ..., instance_targetn ]) ]
  ```

  Example 6. An element i_5 in Figure 1 would be inserted with the following MDDLX command:

  ```
  ALTER DIMENSION DailySales.Product
      ADD INSTANCE i5 INTO LEVEL itemId
      TO LEVELS(brand,category)
      VALUES (b3,c2)
  ```
  ```

- ALTER DIMENSION cube_name.dimension_name
  DELETE INSTANCE instance_value
  FROM LEVEL level_name
- ALTER DIMENSION cube_name.dimension_name
  RECLASSIFY instance_value LEVEL from_level_name
  TO instance_value LEVEL to_level_name
- ALTER DIMENSION cube_name.dimension_name
  SPLIT instance_value LEVEL level_name
  TO new_instance_value_1 [,..., new_instance_value_n]
  FROM LEVELS (level_name_1 [,..,level_name_m])
  USING ROLLUP FUNCTIONS (table_name_1 [,...,table_name_m])
  In the SPLIT operator definition, `new_instance_value_1 [,..., new_instance_value_n]` are the values in which the element represented by `instance_value` will be split, and `table_name_1 [,...,table_name_m]` are `m` tables, one for each level below `level_name` defining the rollup functions for each new element.
- ALTER DIMENSION cube_name.dimension_name
  MERGE instance_value_1 [,..., instance_value_n]
  LEVEL level_name
  TO new_instance_value
- ALTER DIMENSION cube_name.dimension_name
  UPDATE instance_value LEVEL level_name
  TO new_instance_value

In the expressions above, the following data types and properties are supported:

- data_type : DECIMAL | VARCHAR | CHAR | DATETIME
- time_type : DATETIME | DATE | DAY | WEEK | MONTH | YEAR | HOUR
- measure_type : SUM | MIN | MAX | COUNT | AVG
- instance_value, new_ instance_value, instance_target1,... : these are elements in the instance sets of the levels. Thus, their type is the type of the level they belong to.
- axis_specification : dimension_name.level_name
- slicer_specification : dimension_name.level_name = instance_value