Updates, View Maintenance and Time Management in Multidimensional Databases

by

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

avaisman@dc.uba.ar
Universidad de Buenos Aires

Abstract

Updates, View Maintenance and Time Management in Multidimensional Databases

Alejandro A. Vaisman

avaisman@dc.uba.ar

Universidad de Buenos Aires

2001

Usually, OLAP(On Line Analytical Processing) systems provide data visualization through a multidimensional data model according to which a data *fact* is viewed as a mapping from a point in a space of *dimensions* into one or more spaces of *measures*. Moreover, dimensions are organized in levels which conform a hierarchy, providing a way of defining different levels of data aggregation, a central issue in data analysis. In a relational implementation of OLAP(usually called *ROLAP*), we can think of facts as being stored in *fact tables*, while each dimension is described in a *dimension table*. The industry solutions were built under the assumption that data in fact tables reflect the dynamic aspect of the data warehouse, while data in dimension tables represent static information. However, if we think of the data warehouse as a materialized view of data located in multiple sources, it is usual to find situations in which the structure of these sources changes, a new source is added, or an old one dropped. Any of these changes may require updates to the structure of some dimensions. Further, as multidimensional views are designed according to requirements from end users, a redefinition of the initial requirements may cause a dimension update.

In this thesis we argue that accounting for dimension updates is necessary in an OLAP tool in order to avoid constantly rebuilding dimensions from scratch. Thus, we first characterize these updates and study the view maintenance problem when they occur. We developed algorithms which, taking advantage of the nature of the dimension updates, in some cases outperform well-known view maintenance algorithms. We then propose an extension to the MDX language(a standard query language for OLAP) and describe the implementation of TSOLAP, a multidimensional repository which supports dimension updates and view maintenance, developed following the OLE DB for OLAP standard. We discuss the experimental results of tests performed over a real-life case study,

a medical center in Buenos Aires.

In the second part of the thesis we embed our proposal in the temporal database framework, introducing the *Temporal Multidimensional Data Model,* and a temporal query language for OLAP which we called *TOLAP*. *TOLAP* allows expressing complex OLAP queries in an elegant and declarative fashion. We discuss issues like syntax, semantics, safety and expressive power. We also present an implementation including a graphic environment for temporal OLAP. Finally, we show how the temporal approach can be applied to the case study mentioned above.

Acknowledgements

This thesis would not have been possible without the help of many people. Although I have tried not to forget anyone, I apologize in advance for any involuntary omission.

First of all I would like to thank my advisor, Prof. Alberto Mendelzon. I cannot think of this thesis being finished without his guidance and help. I will always be thankful for the time he devoted to my work, the invaluable advice he provided me with, and his patience with my mistakes.

I would also like to thank my counselor at the University of Buenos Aires, Prof. Irene Loiseau, for encouraging me on my work, and for including me in the FOMEC project, which provided funding for my staying in Toronto. Special thanks to Prof. Ken Sevcik at the University of Toronto for his help during the course I took from him. Also thanks to my colleagues Prof. Juan M. Ale, at the University of Buenos Aires, and Mauricio Minuto Espil.

I dedicate this thesis to my mother Berta, my father Manuel, and my sister Nora, for their unconditional love and support.

I wish to thank my friends in Toronto, who made my staying in Canada more enjoyable: Patricia, Gustavo, Flavio and Mariana, and my officemates Davood Rafiei, Attila Barta and George Mihaila. I would especially like to thank Carlos Hurtado, with whom I shared a lot of valuable working time.

Special thanks my friends in Argentina: Daniel, Lidia, Jorge, Anabella, Ernesto and Cecilia.

Also thanks to Sergio Cymerman, Daniel Grippo, Walter Ruaro and Claudio Tirelli for their collaboration with the implementations.

Contents

List of Figures

Chapter 1

Introduction

Since the late seventies, relational database technology has been gaining wide acceptance, to the point that today most organizations rely on it to store their data. However, the needs of these organizations are not the same as they used to be. Increasing market dynamics and competitiveness are leading to the necessity to have the right information at the right time. Managers need to be properly informed in order to take appropriate decisions for running businesses. On the other hand, data possessed by such organizations are usually scattered among different systems, each one devised for a particular kind of job. Traditional systems (usually called OLTP, for On Line Transactional Processing) are not well - suited for these requirements, as they are oriented towards getting the maximum number of transactions per second. As a consequence, new database technologies have emerged in the last five years, namely *Data Warehousing* and *OLAP*(On Line Analytical Processing). They involve architectures, algorithms, tools and techniques for bringing together data from multiple databases or other information sources, into a single repository suited for querying or analysis. This repository is called a *Data Warehouse.* Typical OLAP queries are concerned with historical data, and involve the computation of aggregation and statistical functions. Examples of these kinds of queries are: *"Give me the total sales of toys in Christmas for each of the last five years in California"*, possibly followed by *"Which were the four best-selling toys in the last five Christmas in California?"*.

1.1 OLAP and Materialized Views

1.1.1 OLAP

The OLAP Council White Paper [OLA97] states : "OLAP enables analysts, managers and executives to gain insight into data, through fast, consistent, interactive access to a wide variety of possible views of information." E.F. Codd defined twelve rules which should be accomplished by an OLAP system [CCS93], considering characteristics such as data access, data definition, user requirements and front-end accessibility. According to these rules, users should view data in the way they view their business, data sources should be transparent to the user, and data manipulation must be intuitive and graphic, among other desirable features.

OLAP systems provide data visualization through a multidimensional data model, according to which, a data *fact* is viewed as a mapping from a point in a space of *dimensions* into one or more spaces of *measures.* This multidimensional view of the data is orthogonal to how this data is physically stored: data is perceived by the user as a multidimensional cube where each cell contains a value or measure. According to the actual physical architecture, there are two mainstreams: ROLAP and MOLAP, standing for Relational and Multidimensional OLAP respectively. In ROLAP, data is stored in relational databases, and an intermediate module translates them into a "cube". In MOLAP, data is stored in proprietary arrays, specifically conceived for dimensional analysis.

1.1.2 Materialized Views

A *materialized view* [GM99] is a view such that its tuples are stored in the database, improving query performance, playing the role of a cache which can be directly accessed without looking into the base relations. When the base data is updated, materialized views derived from that data also need to be updated. This process is called *view maintenance.* Computing changes to a view in response to changes to the base relations is called *incremental view maintenance.* This technique avoids recomputing the whole view from scratch.

A view which can be maintained without addressing the base data is called *self-maintainable* [GJM94]. A self-maintainable view can be maintained using only the materialized view and key constraints(which are often available as system's metadata). Formally:

Definition 1 *A view V is self-maintainable with respect to a modification type (insert,delete,update) to a base relation R if for all database states the view can be self-maintained in response to all mod-*

ifications of the indicated type to the base relation R.

Two important results [GJM94], which we will not prove here, are :

- A SPJ (Select-Project-Join) view taking the join of two or more relations is not self-maintainable with respect to insertions.
- A SPJ view is self-maintainable with respect to deletions in R if the key attributes from each occurrence of R in the join are either included in the view, or equated to a constant in the view definition.

When a view V is not self-maintainable in the terms of Definition 1, Quass et al [QGMW96] showed that it is not always necessary to use the complete base relation to maintain such view. In those cases, it is enough to define a set $\mathcal{A}$ of *auxiliary views* which, along with the key and integrity constraints, such that $\{V\} \cup \mathcal{A}$ is self-maintainable. Moreover, they define the notion of *minimal* set of auxiliary views sufficient to maintain the view V.

The subject of materialized views is relevant to our work because a multidimensional database suitable for OLAP involves different sorts of materialized views which pre-compute aggregates in order to speed-up the complex queries often posed to the system. Incremental view maintenance in OLAP has been extensively studied in several papers [LW95, GJM94, GHQ95a, Qua96, QGMW96, GM99, Huy00].

1.2 Multidimensional Modeling

As queries submitted to Data Warehouses are of a different nature than the ones sent to an operational database, the data model supporting the repository should also be different. The Entity - Relationship Model leads to a highly normalized database, which is not suitable for a Data Warehousing environment. Thus, a simpler model, denoted Star Schema [Kim96], is commonly used. In this model, data is organized in two kinds of tables : (a) *fact tables,* which store facts like sales or telephone calls;(b) *dimension tables,* which store data about dimensions (like Customers or Products) through which users analyze facts.

1.2.1 Fact Tables

The main goal in fact table design is to get the smallest possible table without losing information. Fact table characteristics are:

- They capture the interesting elementary transactions of the business in question.
- They are defined over a data retention period(time window for analysis).
- They define the fact granularity. For example, if we do not need to record every transaction, some aggregation must be performed when populating the fact table, storing the daily sales of each product, not every sale made.
- Some facts may not be added across every dimension. For example, it would be usually incorrect to add account balances across time. These facts are called *semi-additive.*
- Fact tables are usually in *Boyce-Codd Normal Form* [Ull88].

Examples of fact table schemas are :

- *Sales(productId, storeId, timeId, qty)*
- *TVwatching(customerId, channelId, timeId, duration)*
- *Balances(customerId, accountNumber, month, amount)*

In the first example, the fact table *Sales* stores a record for each sale of a product identified by *productId* occurred in store *storeId* on instant *timeId.* Analogously, a tuple $< 1221, CNN, 12/12/2000\ 3pm, 1 >$ in fact table *TVwatching* means that customer number 1221 was watching CNN on December 12th, from 3PM to 4PM.

The terms *qty, duration,* and *amount* are the *measures* of the fact table. The other terms are foreign keys referencing attributes in the dimension tables (see below). In a multidimensional view of the data, the measure is the value stored in each cell of the cube.

1.2.2 Dimension Tables

The components of the fact tables described above are the foreign keys of the dimension tables which store the characteristics of the objects represented by such keys. For instance, in the second example of Section 1.2.1, *productId* is a foreign key referencing a dimension table *Product* with a possible schema: (*productId, description, category , brand, company*) storing information about different products. Dimension tables represent the axes of the multidimensional cube. Although dimension tables are usually designed as 1FN relations, when dealing with large dimensions some

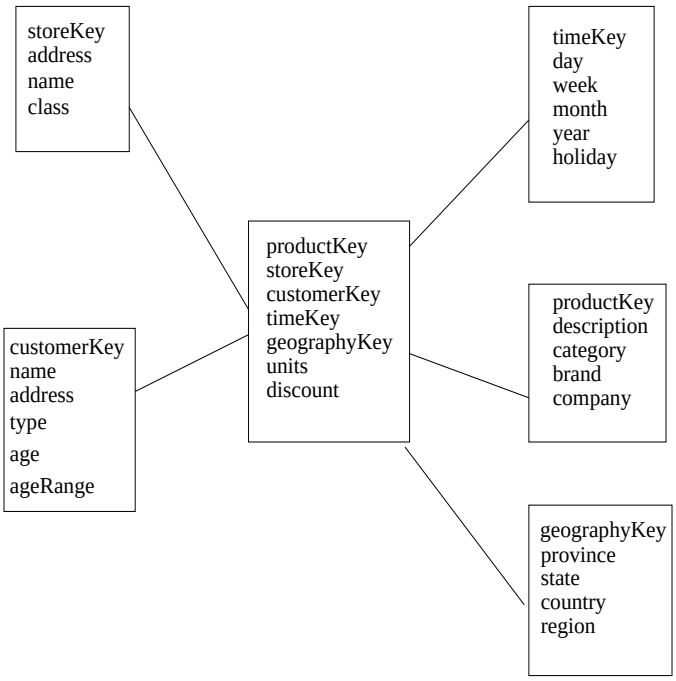

Figure 1.1: A Star Schema for a retail data warehouse.

authors [Sta96] recommend to normalize such tables, in order to reduce the storage cost. This leads to the *Snowflake Schema* [Kim96].

Figure 1.1 shows an example of a Star Schema for a retail data warehouse, with five dimension tables related to a single fact table.

1.2.3 Dimension Hierarchies

In multidimensional modeling, dimensions are usually organized in levels conforming a hierarchy, providing a way of defining different levels of data aggregation. Several models have already been presented [GL96, LW96, AGS+96], and the importance of hierarchies in dimensional modeling has been extensively remarked [JLS99]. In the present work we follow the work of Cabibbo and Torlone [CT97], where dimensions are modeled explicitly through a hierarchy graph and the so-called rollup functions.

Let us introduce the following example illustrating dimension hierarchies. Suppose we have data about policies sold by an insurance company. We may have different kinds of measures (v.g. policies sold, claims information), which can be analyzed through different dimensions as kinds of coverage, customers, salespersons or geographic regions. Let us assume the "Geography" dimension to be

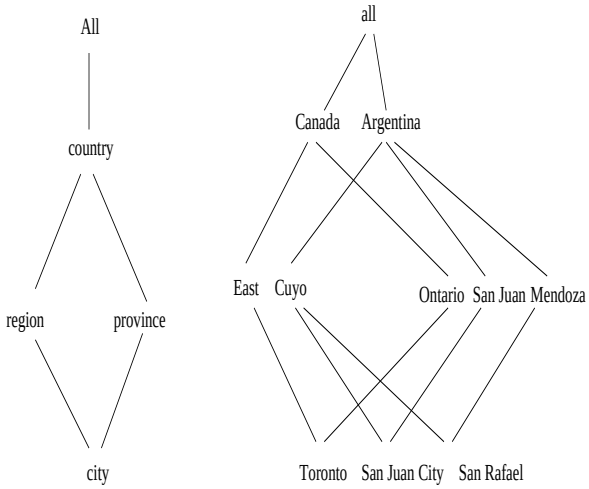

Figure 1.2: A *Geography* dimension

the one in Figure 1.1 (without the "geographyKey" attribute). The dimension's hierarchy could be organized as depicted in Figure 1.2. This figure shows what this classification looks like for three Argentinian and Canadian cities. The hierarchy in Figure 1.2 means that the following functional dependencies hold: $\{city \rightarrow region, city \rightarrow province, province \rightarrow country, region \rightarrow country\}$. In Chapter 2 we will address dimension modeling in depth.

1.3 Data Cubes

Queries to a data warehouse may take a long time to complete due to their complexity and the size of the repository. In order to keep response time within acceptable boundaries, some techniques were developed, mainly making use of view materialization.

1.3.1 The Data Cube Operator

OLAP provides operations allowing users to navigate through data. The most typical ones are *roll-up* and *drill-down*. Roll-up aggregates data at coarser levels while drill-down shows data at finer levels. For instance, let us suppose a data warehouse with three dimensions, $Product$, $Geography$ and $Time$, and a fact table *RegionSales* as follows: *RegionSales(productId, city, month, sales).* Here, *"sales"* is the measure of the fact table.

A typical user will probably want to view data aggregated by *productId*(roll-up operation), and then refine the analysis finding out the monthly sales of each product(drill-down operation).

Moreover, aggregates like moving averages or rating are common in OLAP. The `GROUP BY` operator provided by SQL brings little support to the former operations. Thus, a new operator, called `DATA CUBE` was introduced [GBLP97].

Gray et al define the *Data Cube Operator* as follows [GBLP97]: Given a fact table F with N dimensions, and an aggregate function f, the *Data Cube Operator* builds a table with all possible aggregations over the dimensions in F. The total aggregate is represented by the tuple:

< `ALL,ALL,ALL,..,ALL,f(*)` >

In order to compute the `CUBE` operator, the power set of the aggregation columns must be generated. The table is built taking the union of the 2^N aggregations. If each attribute in the fact table has cardinality C_i, the cardinality of the cube will be given by $\prod(C_i + 1)$.

A data cube for the former data warehouse (and an aggregate function f = `SUM`) would be calculated with the following SQL statements :

```
SELECT productId, city, ''All'', sum(sales)
  FROM RegionSales
  GROUP BY productId, city
UNION
SELECT productId, ''All'', ''ALL'', sum(Sales)
  FROM RegionSales
  GROUP BY poductId
UNION
SELECT ''All'', city, ''ALL'', sum(sales)
  FROM RegionSales
  GROUP BY city
UNION
....
....
UNION
SELECT ''All'', ''All'', ''ALL'', sum(sales)
  FROM RegionSales
```

1.3.2 Aggregate Functions

We will review some basics about aggregate functions, because they will play a central role in the present work. Gray et al [GBLP97] divide aggregate functions in three classes: *distributive, algebraic* and *holistic.*

Definition 2 *Consider aggregating a two-dimensional set of values,* $\{X_{i,j}|i = 1, ..., I; j = 1, ..., J\}$.

- **Distributive** *: An aggregate function* $F()$ *is distributive if there is a function* $G()$ *such that* $F(\{X_{i,j}\}) = G(\{F(\{X_{i,j}|i = 1, \ldots, I\})|j = 1, \ldots, J\})$.

  This definition means that distributive functions can be computed partitioning their input into disjoint sets. Examples of distributive functions are `COUNT, SUM, MAX, MIN`. *For all of them but* `COUNT`, $F = G$ *holds. However, if the statement* `DISTINCT` *is used, like in* `COUNT(DISTINCT)`, *these functions are not distributive. For instance, if we have the set (8,3,5,8,3,3,4,4,7), and* $F =$`SUM`, *partitioning this set into (3,3,4), (5,8,4), (7,3,8), returns a* `COUNT(DISTINCT)` *of 8 (obtained by adding the results from applying* `COUNT(DISTINCT)` *to each subset), while there are only five different values in the set.*

- **algebraic** *: An aggregate function* $F()$ *is algebraic if there is an* $M-$*tuple valued function* $G()$, *and a function* $H()$ *such that*

  $F(\{X_{i,j}\}) = H(\{G(\{X_{i,j}|i = 1, \ldots, I\})|j = 1, \ldots, J\})$

  This definition means that algebraic aggregate functions can be expressed as a scalar function of distributive aggregate functions. The average *function* `AVG` *is an example, where* `AVG = SUM()/COUNT()`.

- **holistic** *: An aggregate function is holistic if there is no constant bound on the size of the storage needed to describe a sub-aggregate. In other words, there is no constant* M *such that an* M*-tuple characterizes the computation* $F(\{X_{i,j}|i = 1, ...I\})$. *Thus, holistic functions cannot be computed by dividing into parts. Median is an example of this kind of function.*

Definition 3 *A set of aggregate functions is self-maintainable if the new value of the function can be computed solely from the old values and from the changes to the base data. Aggregate functions can be self-maintainable with respect to insertions, deletions, or both.*

Some results on maintenance of aggregate functions are [MQM97]:

- Aggregate functions must be distributive in order to be self-maintainable.
- All distributive aggregate functions are self-maintainable with respect to insertions, but not to deletions.
- The function `COUNT(*)` can help to make certain aggregate functions self-maintainable with respect to deletions, by helping to determine when all tuples in a group have been deleted (see Subsection 1.3.3).
- `MAX` and `MIN` are not, and cannot be made, self-maintainable with respect to deletions.

1.3.3 Data Cube Maintenance

A data warehouse will often store a number of materialized views that aggregate data in the fact table, possibly joining this data with one or more dimension tables. In a data warehousing environment, these views are called called *summary tables.* Thus, maintaining a data warehouse becomes a special case of the problem of *view maintenance.* The problem can be characterized in the following way:

1. As data at the sources is added or updated, the summary tables which depend on these data must be also updated. Two options arise : to recompute the summary tables from scratch or to apply incremental view maintenance techniques to avoid such recomputation.
2. While summary tables are maintained, they remain unavailable to the data warehouse users. Thus, the time required for updates must be minimized.

Several works present different solutions [Huy00, MQM97, QW97, LW95, ZGMHW95]. Mumick et al [MQM97] introduced one of the best-known algorithms for efficiently maintaining summary tables in a data warehouse . Thus, we will focus on this algorithm, and in Chapter 2 we will present a variation that improves it in the presence of updates to dimension tables. Another approach, the 2VNL algorithm [LW95], addresses the second problem above, maintaining two versions of the data warehouse in order to have it available 24 hours a day and 365 days a year.

The algorithm developed by Mumick et al, denoted *summary-delta algorithm* applies to *distributive aggregate functions.*

The summary-delta algorithm has two main phases, the *propagate phase*, and the *refresh phase.* The idea is to create a summary-delta table in which the net changes to the summary table due to

changes in the source data are stored. This is performed in the *propagate phase*. Then, during the *refresh phase,* these changes are applied to the summary table.

Let us suppose a fact table $Sales(storeId, itemId, date, qty, price)$, and two dimension tables $Stores(storeId, city, region)$, and $Items(itemId, name, category)$[MQM97].

Let us consider the view $SIDsales$ defined as follows:

```
CREATE VIEW SIDsales AS
SELECT storeId,itemId,date,COUNT(*) AS TotalCount,sum(qty) AS Totalqty
    FROM Sales
    GROUP BY storeId,itemId,date
```

`COUNT(*)` is added in order to make the function `SUM()` self-maintainable with respect to deletions(for instance, if `COUNT(*)` reaches 0, there is no other tuple in the group, so the group can be deleted).

Let $SalesIns$ and $SalesDel$ be tables storing insertions and deletions to the source data. In the propagate phase, a new table(view) is created, where the net changes to the summary tables are stored. This is the aforementioned summary-delta table(in what follows, prefixed as **sd**).

```
CREATE VIEW sdSIDsales (storeId,itemId,date,sdcount,sdqty) AS
SELECT storeId,itemId,date,SUM(_count) AS sdcount,SUM(_qty) AS sdqty
    FROM
         (( SELECT storeId,itemId,date,1 as _count, qty AS _qty
        FROM SalesIns)
         UNION ALL
        (SELECT storeId,itemId,date,-1 as _count,-qty AS _qty
         FROM SalesDel))
GROUP BY storeId,itemId,date
```

During the refresh phase, the net changes, stored in the summary delta table, are applied to the summary table itself. The only case in which base data must be accessed is when `MAX` and `MIN` aggregate functions are involved and a deletion occurs. The refresh algorithm is described below :

```
For each tuple t in sdSIDsales do
      if not exists
        (SELECT * FROM SIDsales d
         WHERE t.storeId = d.storeId AND
         t.itemId = d.itemId AND t.date = d.date)
              then insert t into SIDsales
      else
        (if t.sdcount + d.TotalCount = 0)
              then delete t from SIDsales
              else
              d.TotalCount+= t.sdcount
              d.Totalqty+= t.sdqty)
```

It should be remarked that if the aggregate function is `MAX` or `MIN`, the refresh algorithm recomputes the aggregates from the base data, unless t is deleted or not found, in which case no lookup into the raw data is required.

The main advantage of this approach is that *propagate* can be run without reducing warehouse availability. Only the *refresh* phase must occur within the updating window. Further details on the algorithm can be found in the original paper [MQM97].

1.4 This Thesis

As we explained in Section 1.2, in a relational implementation of OLAP(*ROLAP*) we can think of facts as being stored in *fact tables*, while each dimension is described in a *dimension table*. A common assumption is that data in fact tables reflect the dynamic aspect of the data warehouse, while data in dimension tables represent static information. Furthermore, a data warehouse could be regarded as a materialized view of data located in multiple sources [Wid95]. Thus, it is not difficult to imagine a situation in which the structure of these sources changes, a new source is added, or an old one dropped. Any of these changes may require updates to the structure of some dimensions. Moreover, as multidimensional views are designed according to requirements from end users (a point highly emphasized in many industrial white papers [Inf96, Pil96]), a redefinition of the initial requirements may cause a dimension update. For instance, in Figure 1.2, regions are defined within the same country. A business decision may relax this constraint, allowing regions

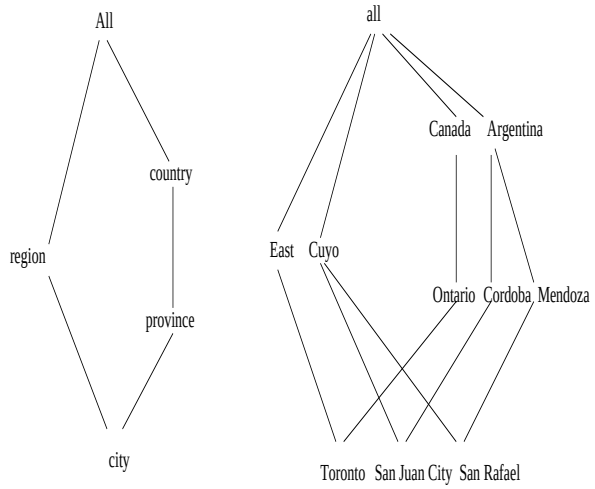

Figure 1.3: Updated *Geography* dimension

to be spread across different countries. This may be represented in the lattice of Figure 1.2 by deleting the edge joining the *region* and *country* levels, and adding a new edge from *region* to the distinguished level *All.* Figure 1.3 shows the resulting dimension. Additionally, new salespersons may be hired or fired, new kinds of coverage introduced or discontinued, regions may be reorganized, merged or split, etc.

There is, then, a wide range of possible dimension updates, which the existing models fail to capture. Kimball analyzes this problem [Kim96] to some degree, giving rise to the concept of *slowly changing dimensions*, partially covering updates to dimension instances. Algorithms that perform view maintenance, like the Summary-Delta method, focus on updates to fact tables, and must be modified in order to be efficiently applied when a dimension update occurs.

Let us consider again the *Geography* dimension of figures 1.2 and 1.3. A user may be interested in querying the multidimensional database as of the instant depicted in Figure 1.2. Moreover, since the schemas of the fact tables are composed of attributes from associated dimensions, certain updates may trigger schema evolution over such fact tables. Suppose we wish to collect data at a granularity level finer than *city*, for instance *neighborhood.* Any fact table associated with the *Geography* dimension would require its schema to be updated. We argue in this thesis that in an evolving scenario like this, OLAP systems need temporal features to keep track of the different states of a data warehouse throughout its lifespan.

1.4.1 Our Approach

We believe it is clear that dimension updates must be addressed by multidimensional models. Our proposal develops in incremental steps as follows:

- we present a complete characterization of the possible dimension updates in a multidimensional model, and define a collection of operators which perform them.
- We study the effect of these updates over a class of materialized views over the dimension levels, and give algorithms for efficiently maintaining those views, which turn out to be more efficient than the well-known *summary-delta method* when dealing with dimension updates. We also present an implementation of the model, which extends Microsoft's MDX proposal [Mic98], and apply it to a real-life case study (a medical center in Buenos Aires).
- We then introduce a *temporal multidimensional data model* and a temporal query language supporting it, which we called *TOLAP* (*Temporal OLAP*), combining some of the temporal features of query languages like TSQL2 or SQL/TP [Sno95, Tom97] with some of the high-order features of languages like HiLog or SchemaLog [CKW89, LSS93, LSS97].
- Finally, we present an implementation of the temporal model on top of an ORACLE database. We also analyze the medical center case study using the temporal approach.

1.4.2 Thesis Organization

The organization of this thesis is as follows:

Chapter 2 introduces a multidimensional model which accounts for dimension updates. We present the set of update operators, as well as algorithms for incrementally updating the data cube.

In Chapter 3 we present a relational implementation of the model discussed in Chapter 2, extending the MDX language developed by Microsoft. We discuss different representation alternatives, and describe the implementation of a multidimensional data provider which we called TSOLAP, built using the OLEDB for OLAP standard. In Chapter 4 we present experimental results through a case study in which we used real medical data taken from a health-care center in Argentina.

Chapter 5 discusses the need for a temporal model for OLAP. We argue that in the presence of dimension updates which trigger changes on the granularity of the facts, a temporal model supporting schema evolution is required. We discuss existing temporal data models and show that they are not suitable for these requirements, and extend the model presented in Chapter 2.

In Chapter 6 we introduce a temporal query language called *TOLAP* supporting the temporal model. Chapter 7 shows an implementation of the model we present in Chapter 5. A major feature of this implementation is a visualizer allowing schema and instance browsing across time. A *TOLAP* implementation is described, and experimental results are presented using the medical center case study we introduced in Chapter 3.

Finally, in Chapter 8 we summarize the dissertation, highlighting our contributions and discussing possible future research directions.

Chapter 2

Multidimensional Updates and View Maintenance

2.1 Introduction

Dimensions represent the framework within which factual data is summarized for analysis. Therefore, changes in analysis requirements and/or in the structure of the data sources almost always imply changes in the dimensions of the model. In Chapter 1 we showed that these changes are not limited to the addition or deletion of tuples, but they may also involve the hierarchical structure according to which dimensions are organized. All these kinds of dimension updates are poorly supported (or not supported at all) in current commercial systems. In this chapter we introduce the concept of *dimension update,* and propose a framework for its analysis. We also present a set of operators performing structural and instance updates over dimensions, and algorithms for data cube maintenance in a ROLAP environment.

We commented in Chapter 1 that Mumick et al [MQM97] proposed the *summary-delta algorithm* for incremental maintenance of a set of materialized aggregate views defined over the same base table. They apply this approach to *generalized cube views*, and addresses maintenance under updates to fact tables, with a brief discussion of instance updates to dimension tables. In this chapter we present an algorithm which improves the summary-delta algorithm by avoiding, whenever possible, joins and aggregate computations.

This chapter is organized as follows: in the next section we introduce a multidimensional model supporting updates to a dimension's instance and structural updates to a dimension's hierarchy.

In Section 2.3 a collection of primitive operators that perform these updates is defined. In Section 2.4 we study the effect of these updates over a class of materialized views defined over dimension levels, and present an algorithm to maintain them, which turns out to be more efficient than the *summary-delta algorithm* when dealing with dimension updates. Finally, in Section 2.5 we introduce complex operators which encapsulate series of primitive operations.

Note Part of what we present here can be found in previous work [HMV99a, HMV99b].

2.2 Multidimensional Model

Cabibbo and Torlone [CT97] introduced a multidimensional model in which dimensions are organized as hierarchies of levels corresponding to possible granularities of data, and rollup functions describing how data are related within those hierarchies. Factual data are represented in this model through *f-tables.* The authors also propose a multidimensional calculus. We chose this model as a starting point for our work, among several other multidimensional models which had been introduced in the last five years [GL96, LW96, AGS+96]. Our model, however, defines a set of constraints not present in the model of Cabibbo and Torlone, and makes a clear distinction between schema and instances of the multidimensional database objects, as it is usual in relational databases.

We already remarked that dimensions can be organized into hierarchies which enable the definition of different levels of data aggregation. We argue that the usual assumption in relational OLAP implementations, regarding fact tables as dynamic and dimension tables as static, turns out to be not true in most practical cases. Changes in the data will often require updates to the dimension tables. In the case study we will present in Chapter 4, we will show that in a Health Care clinic, new doctors may be hired or fired, medical practices can be opened or closed, different kinds of health insurance plans introduced or discontinued, etc. Structural updates could also occur, as categorization levels are added or deleted. For example, doctors may cease to be classified according to salary ranges. In this section we will introduce a model accounting for these situations.

2.2.1 Dimensions and Fact Tables

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

Definition 4 (Dimension Schema) *A* **dimension schema** *is a tuple* $(dname, L, \preceq)$ *where:*

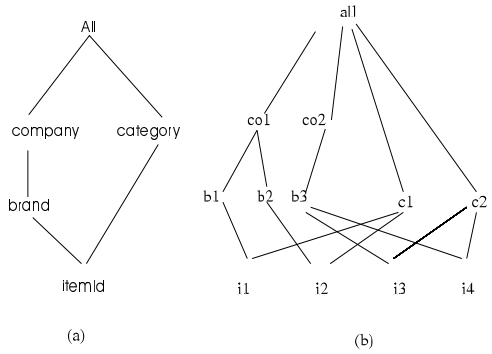

Figure 2.1: (a) A dimension schema. (b) A dimension instance.

- *dname* $\in \mathbf{D}$ *is the name of the dimension.*
- $L \subseteq \mathbf{L}$ *is a finite set of levels, which contains a distinguished level name All, such that* $dom(All) = \{all\}$.
- $\preceq$ *is a relation over levels, such that* $\preceq^*$*, its transitive and reflexive closure, is a partial order, with a unique bottom level, called* l_{inf}*, a unique top level, "All", and, for every level* $l \in L$*,* $l_{inf} \preceq^* l$ *and* $l \preceq^* All$ *hold. Moreover, if* l_a *and* l_b *are levels in* L*, and* $l_a \preceq^* l_b$*, then* $l_a \not\preceq l_b$.

Definition 5 (Dimension Instance) *A* **dimension instance** *is a tuple* (D, ρ) *where:*

- D *is a dimension schema.*
- ρ *is a set of partial functions such that:*
  - *for each pair of levels* l_1*,* l_2 *such that* $l_1 \preceq l_2$*, there exists a rollup function (partial function)* $\rho_{l_1}^{l_2} : dom(l_1) \rightarrow dom(l_2)$.
  - *for each pair of paths* τ_1, τ_2*, in the graph with nodes in* L *and edges in* $\preceq$*,* $\tau_1 =< l_1, l_2, \ldots, l_{n-1}, l_n >$*, and* $\tau_2 =< l_1, l'_2, \ldots, l'_{m-1}, l_m >$*, such that* $l_n = l_m$*, we have* $\rho_{l_1}^{l_2} \circ \ldots \circ \rho_{l_{n-1}}^{l_n} = \rho_{l'_1}^{l'_2} \circ \ldots \circ \rho_{l'_{m-1}}^{l'_m}$.

    We denote this property Consistency.
  - *for each triple of levels* $l_1, l_2, l_3 \in L$ *such that* $l_1 \preceq l_2$ *and* $l_2 \preceq l_3$*,* $ran(\rho_{l_1}^{l_2}) \subseteq dom(\rho_{l_2}^{l_3})$.

From Definition 5 we infer that given a dimension instance, for each triple of levels l, l' and l'', such that $l \preceq l'$ and $l \preceq l''$, $dom(\rho_l^{l'}) = dom(\rho_l^{l''})$.

Example 1 *Figure 2.1 shows an example of a simple dimension schema and instance for a dimension* Product. *Here, for the dimension schema, dname = Product, L = {itemId, brand, company, category, All}. Relation $\preceq$ contains the following pairs: $itemId \preceq brand, brand \preceq company, company \preceq All, itemId \preceq category, category \preceq All$.*

Notation In what follows, dimension will stand for dimension instance, except when noted. It must be noticed that each dimension level can be described by level attributes. However, in order to enhance clarity, we omit attributes at this time. We will consider level attributes in the next chapters.

Given a dimension level l, the *instance set* of l, denoted $instset(l)$, is the set containing the elements in l.

Example 2 *For the dimension instance Product given in Example 1 and Figure 2.1, the instance set of level ItemId, is $\{i_1, i_2, i_3, i_4\}$.*

Definition 6 (Transitive Closure of a Set of Rollup Functions) *The transitive closure of a set of rollup functions ρ, denoted ρ^*, is the set that contains a rollup function for each pair of levels $l_m, l_n \in L$, $l_m \preceq^* l_n$, such that if $l_m = l_n$, $\rho^{*l_n}_{l_m} = identity$; otherwise, if $< l_m \ldots l_n >$ is a path from l_m to l_n in the graph with nodes in L and edges in $\preceq$, $\rho^{*l_n}_{l_m} = \rho^{l_{m+1}}_{l_m} \circ \ldots \circ \rho^{l_n}_{l_{n-1}}$.*

Definition 7 (Fact Table) *A* **fact table schema** *is a tuple $s = (fname, Lset, m)$, where $fname \in \mathbf{F}$ is a fact table name, Lset is a set of levels, and m is a level, called the* measure *of the fact table. Moreover, given a fact table schema $(fname, Lset, m)$, a* **point** *is a mapping from each level l_i in Lset to a value in $dom(l_i)$. Given a fact table schema $s = (fname, Lset, m)$, a* **fact table instance** *over it is a partial function which maps points of s to elements in $dom(m)$.*

Definition 8 (Multidimensional Database) *A* **multidimensional database schema** *is a pair $MS = (DS, FS)$, where DS is a set of dimension schemas, and FS is a set of fact table schemas. A* **multidimensional database instance** *is a tuple $MD = (D, F)$, where D is a set of dimensions, and F a set of fact tables instances.*

Notation A *dimension set* is a set of dimensions. Given a dimension set D, a *level group* is a set of levels containing exactly one level for each dimension in $D \setminus \{Measure\}$, where $Measure$ is a distinguished dimension. Aggregation takes place over this distinguished dimension (see Subsection

2.2.2). GB_D will denote the *set of all the possible level groups,* and $GBottom_D$ a level group containing the bottom levels of each dimension in $D \setminus \{Measure\}$. A *base fact table* is a fact table with schema $(fname, GBottom_D, m)$.

2.2.2 Data Cubes

Several classes of aggregate views have been used to fulfill different requirements in OLAP systems. We commented in Section 1.3 that Gray et al [GBLP97] introduced the *data cube operator* as the union of a set of cube views that contains data from a base fact table, aggregated over all the possible groups of attributes in it. We will extend the data cube operator in order to include views computing aggregates over the levels of the dimensions and define the *CubeView* operator to express them. Moreover, we will not require a unique view holding the union of all the possible ones. Aggregation takes place over a level m in the distinguished dimension $Measure$. We also define an *aggregate function* Ag over m as a function with signature $Ag : 2^{dom(m)} \rightarrow dom(m)$.

Definition 9 (Cube View) *Given a set of dimensions D, a base fact table f_{base}, with measure m, and a level group $GB = \{l'_1, \ldots l'_n\}$, $CubeView(f_{base}, D, GB)$ yields a fact table f, with* schema *$s = (fname, GB, m)$, and* instance *defined as follows:*

- *given a point c, $S(c)$ is the set that contains all points c' in the domain of f_{base}, such that for each pair of levels $l_1 \in GBottom_D$ and $l_2 \in GB$, belonging to the same dimension, $c(l_2) = \rho^{l_2}_{l_1}(c'(l_1))$ holds, where $\rho^{l_2}_{l_1}$ is in the set of rollup functions ρ^* of that dimension.*
- *for the points c such that $S(c) \neq \phi$, we have $f(c) = Ag(S(c))$.*

Notice that m and Ag are not parameters of the CubeView, *because they can be inferred from the context(see Section 3.4).*

In what follows we will assume that Ag is the function SUM. This can be easily generalized to the other four basic SQL aggregate functions.

Example 3 *Consider a set of dimensions* D= {Product,Store,Time,Sales}. *The schema and rollup functions for* Product *are given in figure 2.1. Schemas for* Time *and* Store *follow from the rollup functions given below . We defined the following rollup functions: $\rho^{region}_{storeId} = \{s_1 \mapsto r_1, s_2 \mapsto r_2, s_3 \mapsto r_3\}$; $\rho^{week}_{day} = \{d_1 \mapsto w_1, d_2 \mapsto w_1, d_3 \mapsto w_2, d_4 \mapsto w_2\}$ (we have omitted the rollups to* All*). Let us consider the following 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 fact table* $CubeView(DailySales, D, \{itemId, storeId,$week$\})$*, call it* Sales_ISW*, is the following:*

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

In terms of the model introduced here, the *Data Cube Operator*, can be seen as a set of cube views defined over a set of level groups, as the following definition shows.

Definition 10 (Data Cube Operator) *Given a dimension set D, a base fact table f_{base} with measure m, and a set of level groups $GBSET$, the data cube operator $DataCube(D, f_{base}, GBSET)$, gives a multidimensional database $(D, F \cup \{f_{base}\})$, such that for every level group GB in $GBSET$ there is one and only one fact table f in F, where $f = CubeView(f_{base}, D, GB)$.*

Note that this is not the same data cube operator defined by Gray et al [GBLP97]. They define the operator as the union of 2^N aggregate views computed over the N attributes of the base fact table(see Subsection 1.3.1). The data cube operator of Definition 10 generates a *set of views* over all the level groups in the set of level groups $GBSET$ which the operator takes as an argument.

Example 4 *For the dimension set D and the base fact table* Daily_Sales *of Example 3, we want to materialize views over the following set of level groups: $GBSET = \{\{itemId,storeId, day, sales\}$, $\{itemId, storeId, week, sales\}$, $\{brand, storeId, week, sales\}$, $\{brand,All,All,sales\}\}$. $DataCube(D, f_{base}, GBSET)$ will contain a base fact table* Daily_sales*, and four aggregate views, one for each level group in $GBSET$.*

2.3 Dimension Updates

We will define a basic set of operators that will allow modifying either the schema or an instance of a given dimension, classifying them into two subsets :

- Structural Update Operators
- Instance Update Operators.

Operators in the first set modify the structure of a dimension. Operators in the second set modify the schema and the instance of a dimension.

2.3.1 Structural Update Operators

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

Operator 1 (Generalize) *Given a dimension* $d = ((dname, L, \preceq), \rho)$*, two levels,* $l \in L$*,* $l_n \notin L$*, and a function* $f_l^{l_n} : instset(l) \rightarrow dom(l_n)$*,* Generalize$(d, l, l_n, f_l^{l_n})$ *is a new dimension* $((dname, L \cup \{l_n\}, \preceq \cup\{(l, l_n), (l_n, All)\} \setminus (l, All)), \rho')$*, where* ρ' *is the set containing the rollup functions* ${\rho'}_{l_i}^{l_j}$*, such that:*

- ${\rho'}_l^{l_n} = f_l^{l_n}$;
- ${\rho'}_{l_n}^{All} = \{(e, all) \mid e \in ran(f_l^{l_n})\}$;
- ${\rho'}_{l_i}^{l_j} = \rho_{l_i}^{l_j}$*, for all other levels* l_i, l_j.

Example 5 *Suppose we want to define a new level in the* Store *dimension of Example 3, for instance the type of store, with two possible values,* t_1 *and* t_2*. The operation would be defined as:* Generalize(Store, storeId, storeType, $f_{storeId}^{storeType}$)*, where* $f_{storeId}^{storeType} = \{(s_1, t_1), (s_2, t_1), (s_3, t_2)\}$*. Figures 2.2(a) and (b) show the structural and instance modifications triggered by the operator.*

Notation We say that two levels are independent, denoted $l_a \parallel l_b$, when $l_a \not\preceq^* l_b$ and $l_b \not\preceq^* l_a$.

The *Relate* operator links two independent levels in a dimension. A precondition for *Relate* is the existence of a function f between the instance sets of the levels being related, such that the dimension instance remains consistent. We call this function a *consistency function.*

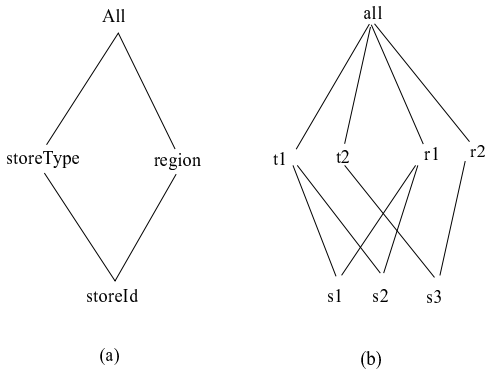

Figure 2.2: (a)Schema after Generalization (b) Instance after Generalization.

Definition 11 (Consistency Function) *Given a dimension d, and two independent levels l_a and l_b in d, a* consistency function $f_{l_a}^{l_b}$ *between instset(l_a) and instset(l_b) is a function defined as follows:*

$f_{l_a}^{l_b} = \{i_a \mapsto i_b \mid \exists i \in l_{inf}, \rho^{*l_a}_{l_{inf}}(i) = i_a,\ \rho^{*l_b}_{l_{inf}}(i) = i_b\}$

Example 6 *In Figure 1.2, we cannot relate level* region *to level* province *because we cannot map two regions to three provinces. However, it is possible to relate* province *to* region*, as we can map* San Juan *and* Mendoza *to Cuyo and* Ontario *to East.*

When conditions for relating two levels l_a and l_b are met, we must delete all the redundant rollup functions that may appear, as the model only includes direct rollups. For instance, in Figure 1.2 we must delete the rollup functions between levels *city* and *region* when relating *province* and *region*.

Operator 2 (Relate Levels) *Given a dimension, say $d = ((dname, L, \preceq), \rho)$, a pair of levels $l_a \in L$, and $l_b \in L$, such that $l_a \parallel l_b$, and a consistency function $f_{l_a}^{l_b}$;* Relate(d, l_a, l_b) *is the dimension $((dname, L, \preceq'), \rho')$, where:*

- $\preceq' = \preceq \cup \{(l_a, l_b)\} \setminus \{(l_i, l_b) | l_i \preceq^* l_a \wedge l_i \preceq l_b\} \setminus \{(l_a, l_k) | l_a \preceq l_k \wedge l_b \preceq^* l_k\}$;
- $\rho'^{l_b}_{l_a} = f_{l_a}^{l_b}$;
- $\rho'^{l_j}_{l_i} = \rho^{l_j}_{l_i}$, *for all other levels l_i, l_j.*

Note that we do not need to specify $f_{l_a}^{l_b}$ in the definition of the operator, because it is uniquely determined by the pre-existing rollup functions.

The *Unrelate* operator deletes a relation $\preceq$ between two levels l_a and l_b, such that $l_a \preceq l_b$. The operator must guarantee that levels below l_a in the hierarchy, will still be able to reach the same

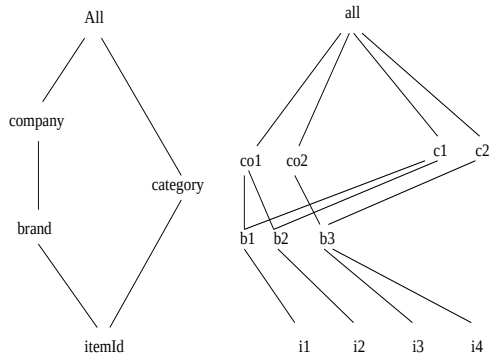

Figure 2.3: Dimension Product after *Relate(Product,brand,category)*.

levels they reached before the unrelate operation. For instance, if $l_a \preceq l_b$ and $l_b \preceq l_c$ hold, we must preserve $l_a \preceq l_c$ by making it explicit in case we delete $l_a \preceq l_b$. Notice that l_b cannot be "All" and l_a cannot be the dimension's bottom level, unless levels parallel to (i.e. independent from) l_b and l_a exist. These conditions are necessary in order to guarantee a graph with a unique source and a unique sink.

Operator 3 (Unrelate Levels) *Given a dimension* $d = ((dname, L, \preceq), \rho)$, *and two levels,* $l_a \in L$ *and* $l_b \in L$, $l_a \neq l_{inf}$ $l_b \neq All$, *such that* $l_a \preceq l_b$, Unrelate(d, l_a, l_b) *is a new dimension* $((dname, L, \preceq'), \rho')$, *where:*

- $\preceq' = \preceq \setminus \{(l_a, l_b)\} \cup \{(l_i, l_b) | l_i \preceq l_a \wedge l_i \not\preceq^*_{ab} l_b\} \cup \{(l_a, l_j) | l_b \preceq l_j \wedge l_a \not\preceq^*_{ab} l_j\}$ [1];
- ${\rho'}^{l_j}_{l_a} = \rho^{l_b}_{l_a} \circ \rho^{l_j}_{l_b}$, *if* $l_b \preceq l_j$;
- ${\rho'}^{l_b}_{l_i} = \rho^{l_a}_{l_i} \circ \rho^{l_b}_{l_a}$, *if* $l_i \preceq l_a$;
- ${\rho'}^{l_j}_{l_i} = \rho^{l_j}_{l_i}$, *for all other levels* l_i, l_j.

If $l_a = l_{inf}$ *or If* $l_b = All$, *the following holds:*

- Unrelate(d, l_{inf}, l_b) = Relate(d, l_c, l_b), *where* l_c *is the level closest to* l_a *such that* $l_b \parallel l_c$ *and* Relate(d, l_c, l_b), *is possible.*

  By closest level we mean that there is no path $< l_{inf}, \ldots l_j \ldots l_c >$ *such that* Relate(d, l_j, l_b), *is possible. If there exist two such levels within the same distance from* l_{inf}, *anyone can be chosen.*

[1] The last conditions prevent the addition of the arcs in case alternative paths between (l_i, l_b) or (l_a, l_j) (that is, paths not including $l_a l_b$ as a subpath) existed in $\preceq$. The expression $l_i \preceq^*_{ab} l_j$ means that there is a path between l_i and l_j including the edge $l_a l_b$.

- Unrelate(d, l_a, All) = Relate(d, l_a, l_d), *where* l_d *is the level closest to All such that* $l_d \parallel l_a$ *and* Relate(d, l_a, l_d), *is possible.*

  By closest level we mean that there is no path $< l_d, \ldots l_j \ldots All >$ *such that* Relate(d, l_a, l_j), *is possible. If there exist two such levels within the same distance from* l_{inf}, *anyone can be chosen.*

Example 7 *Figure 2.3 shows the result of applying operator* Relate(Product, brand, category), *to the dimension of Figure 2.1. Notice that relation* itemId $\preceq$ category *was deleted in order to avoid redundant edges. Unrelating* brand *and* category *will take the dimension back to its previous state(i.e., the state before* Relate(Product, brand, category) *occurred). Note that we could not unrelate* itemId *and* brand, *because this would not yield a valid dimension.*

The *DelLevel* operator deletes a level and its rollup functions. The level to be deleted cannot be the lowest one in the dimension (l_{inf}), unless it rolls up to only one level above it(for instance, we could delete level "hour" in the schema corresponding to the instance of figure 2.7, but we could not delete level *itemId* in figure 2.1). As it was the case with the *Relate* operator, taking into account that we only define the direct rollups, when deleting a level we must add the functions between levels above and below it.

Operator 4 (Delete Level) *Given a dimension* $d = ((dname, L, \preceq), \rho)$ *and a level* $l \in L, l \neq All$,*such that, if* $l = l_{inf}$, *there is only one level* l_j *such that* $l \preceq l_j$, DelLevel(d, l) *is a new dimension* $((dname, L' = L \setminus \{l\}, \preceq'), \rho')$, *such that:*

- $\preceq' = \preceq \setminus \{(l_1, l_2) \mid (l_1 = l) \vee (l_2 = l)\} \bigcup \{(l_1, l_2) \mid (l_1 \preceq l) \wedge (l \preceq l_2) \wedge (l_1 \not\preceq^* l_2\}$[2];
- ${\rho'}_{l_i}^{l_j} = \rho_{l_i}^{l} \circ \rho_{l}^{l_j}$, *if* $l_i \preceq l$ *and* $l \preceq l_j$;
- ${\rho'}_{l_i}^{l_j} = \rho_{l_i}^{l_j}$ *for all other levels* l_i, l_j.

Example 8 *If we delete level* brand *from the* Product *dimension, we would obtain the schema and instance depicted in Figure 2.4.*

[2]this last condition, like in the previous operator, implies that no alternative paths between (l_1, l_2) exist.

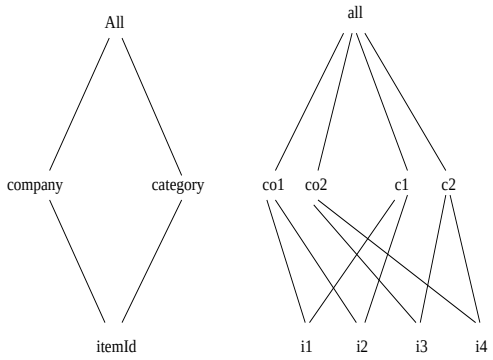

Figure 2.4: An example of *DelLevel*.

2.3.2 Instance Update Operators

The following operators add or delete instances to and from a level in a dimension. They also impose some constraints on the way these updates can be performed. In simple words, elements are inserted into levels "from top to bottom", and are deleted "from bottom to top".

The *AddInstance* operator inserts a new element, say x, into a level l_a(i.e., an element not belonging to the instance set of l_a). We must provide the operator with all the pairs (l_i, x_i), such that every l_i is a level to which l_a directly rolls up($l_a \preceq l_i$), and x_i is the element in the instance set of l_i to which x will roll up (i.e. $\rho_{l_a}^{l_i}(x) = x_i$).

Operator 5 (Add Instance) *Given a dimension* $d=((dname, L, \preceq), \rho)$*, a level* l_a*, an element* $x_a \in dom(l_a)$*,* $x_a \notin instset(l_a)$*, and a set of pairs* $P = \{(l_1, x_1), \ldots (l_n, x_n)\}$*, where* l_i *is a level, and* x_i *is an element in* $instset(l_i)$*, and the following hold:*

- $dom(P) = \{l_i \mid l_a \preceq l_i\}$;
- *for each pair* $\{(l_i, x_i), (l_j, x_j)\} \in P$*, if there is a level* $l \in L$ *such that* $l_i \preceq l$ *and* $l_j \preceq l$*, then* $\rho^{*l}{}_{l_i}(x_i) = \rho^{*l}{}_{l_j}(x_j)$;

AddInstance(d, l, x_a, P) *is a new dimension* $((dname, L, \preceq), \rho')$ *where:*

- ρ' *is the set containing the rollup functions such that* $\rho'^{l_j}_{l_a} = \rho^{l_j}_{l_a} \cup \{(x_a, x)\}$*, for each* $(l_j, x) \in P$;
- $\rho'^{l_j}_{l_i} = \rho^{l_j}_{l_i}$*, for all other levels* l_i, l_j*;*
- $instset'(l_a) = instset(l_a) \cup \{x_a\}$.

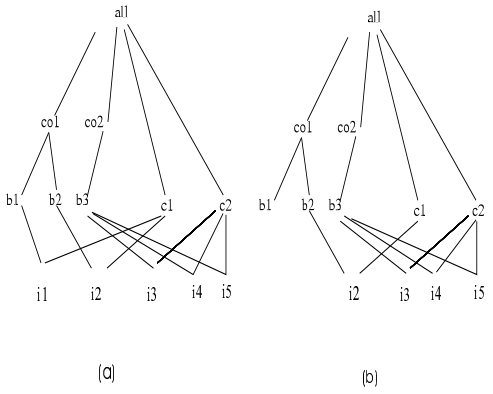

Figure 2.5: (a) Add Instance (b) Delete Instance.

The *DelInstance* operator deletes an element belonging to the instance set of a level l_a. It is only defined when no element of any level l_i such that $l_i \preceq l_a$, rolls up to the element being deleted.

Operator 6 (Delete Instance) *Given a dimension* $d = ((dname, L, \preceq), \rho)$*, a level* $l_a \in L$*, and an element* $x_a \in instset(l_a), x_a \notin \bigcup_{l \in L} ran(\rho_l^{l_a})$*,* DelInstance$(d, l_a, x_a)$ *is a new dimension* $((dname, L, \preceq), \rho',)$ *where:*

- ρ' *is the set containing the rollup functions such that* ${\rho'}_{l_a}^{l_j} = \rho_{l_a}^{l_j} \setminus \{(x_a, \rho_{l_a}^{l_j}(x_a))\}$;
- ${\rho'}_{l_i}^{l_j} = \rho_{l_i}^{l_j}$*, for all other levels* l_i, l_j;
- $instset'(l_a) = instset(l_a) \setminus \{x_a\}$.

Example 9 *In order to add a new item, say* i_5*, to the* Product *dimension of figure 2.1, we should apply:* AddInstance (Product, itemId, $i_5, \{(brand, b_3), (category, c_2)\}$)*. Figure 2.5(a) shows the result. After that, we delete* i_1 *from level* itemId*. Note that in this case, we can only delete an element belonging to* instset(itemId)*, due to the operator preconditions. The operation is invoked as* DelInstance(Product, itemId, i_1)*. After this, it would be possible to delete* b_1*, because it is a leaf in the graph. See figure 2.5(b).*

Theorem 1 (Correctness of Update Operators) *Operators 1 to 6 are correctly defined. That is, given a dimension* $d = (D, \rho)$*, and an update operator* θ *such that* $\theta(d) = d'$*,* d' *is a dimension satisfying Definitions 4 and 5.*

Proof Theorem 1 *For the sake of simplicity, we will use the following additional definitions :*

Given a graph $G = (L, \preceq)$, representing a Dimension Schema $(dname, L, \preceq)$, a path $\tau =< l_1 l_2 \ldots l_{n-1} l_n >$ in it, and a rollup function ρ associated with each pair of levels $l_i l_{i+1}$, we define the path composition with respect to this function as $O(\tau) = \rho_{l_1}^{l_2} o \ldots o \rho_{l_{n-1}}^{l_n}$.

Then, we can express the second condition of definition 5 (consistency property) in the following way: for each pair of paths $\tau =< l_1 \ldots l_j >$ and $\tau_1 =< l_1 \ldots l_m >$ in the graph, such that $l_j = l_m$, $O(\tau) = O(\tau_1)$ holds.

We must prove correctness for each one of the operators defined in section 3. We will omit the proofs of properties that follow straightforwardly from the operator's definition.

Generalize *We will show that consistency is satisfied by every ${\rho'}_{l_i}^{l_j}$ in ρ'.*

By definition, every new path in the graph must include l, l_n, and All. *Thus, as $f_l^{l_n}$ is a function, ${\rho'}_l^{All} = \{(e, all) \mid e \in ran(f_l^{l_n})\}$, and l_n has no incoming arcs except the one from l, no new path could violate consistency, because for every new path τ in G, $O(\tau) = all$ holds.*

Now, we must prove the third property in definition 4. We must show that $\preceq'$ is a partial order. We will do this by analyzing which pairs of levels, of the form (l_i, l_j) are added to or deleted from $\preceq^*$.

By the operator's definition, if $(l_i, l_j) \notin \preceq^*$, then, $(l_i, l_j) \notin \preceq'^*$. On the other hand the only pairs in $\preceq'^*$, not in $\preceq^*$ are (l, l_n), (l_n, All), and (l_n, l_n), while the only pair which may be deleted is (l, All). Thus, as $\preceq^*$ was anti-symmetric, so is $\preceq'^*$, because no tuple added can prevent that (no (l_n, l_i) pair is added unless $l_i = All$ or $l_i = l_n$). Moreover, adding (l_n, l_n) guarantees reflexiveness.

Finally, if $(L, \preceq)$ was a lattice, so is $(L, \preceq')$. This follows from the fact that, by the definition of $\preceq'$, every level that reached All *through l in $< L, \preceq >$, will do the same through l_n in $< L', \preceq' >$, and no sink or source are added to G.*

Relate Levels *A precondition for this operator is the existence of a consistency function. Thus, as no rollup function is added, consistency is preserved.*

We proceed now with the second part of the proof.

When l_a is related to l_b, and because $l_a \not\preceq^* l_b$ and $l_b \not\preceq^* l_a$, the only pairs in $\preceq'^*$ and not in $\preceq^*$ (besides (l_a, l_b)), are of the form:

- *(l_i, l_b), if $l_i \not\preceq^* l_b \wedge l_i \preceq^* l_a$;*

- (l_a, l_j), *if* $l_a \not\preceq^* l_j \wedge l_b \preceq^* l_j$;
- (l_i, l_j), *if* $l_i \not\preceq^* l_j, l_i \preceq^* l_a \wedge l_b \preceq^* l_j$;

  Note that some pairs (l_i, l_j) *may already be in the original dimension, if a path not including an* $l_a l_b$ *subpath existed between* l_i *and* l_j.

On the other hand, no pair is deleted from $\preceq^*$.

From above, it follows that no cycle could be induced by the new pairs. Thus, $\preceq'^*$ *is a partial order.*

Unrelate Levels *We will first prove that* $O(\tau_1) = O(\tau_2)$, *for every pair of paths* τ_1 *and* τ_2 *with the same extreme nodes, still holds after unrelating two nodes.*

By the operator's definition, we know that $\rho'^{l_j}_{l_i} = \rho^{l_j}_{l_i}$, *for all pairs of levels* l_i, l_j *such that* $l_a \notin \{l_i, l_j\}$ *and* $l_b \notin \{l_i, l_j\}$. *In any other case, the rollup functions* $\rho^{l_a}_{l_i}$ *and* $\rho^{l_j}_{l_b}$, *such that* l_i *and* l_j *are levels directly below* l_a *and above* l_b *respectively, are composed with* $\rho^{l_b}_{l_a}$. *Let us analyze the case such that* $l_i \preceq^* l_a$ *and* $l_b \preceq^* l_j$. *No path from* l_i *to* l_j *in the new graph will contain* $l_a l_b$ *as a subpath, but every path from* l_i *to* l_j *in* $\preceq$ *will be associated to at least one path from* l_i *to* l_j *in* $\preceq'$. *If the original paths verified* $O(\tau_1) = O(\tau_2)$, *the new ones will also satisfy this condition, because by definition, if in the expression* $O(\tau) = \rho^{l_2}_{l_i} o \ldots \rho^{l_b}_{l_a} o \rho^{l_t}_{l_b} o \ldots \rho^{l_j}_{l_s}$ *we replace* $\rho^{l_b}_{l_a} o \rho^{l_t}_{l_b}$ *with* $\rho'^{l_t}_{l_a}$, *we have that* $O(\tau_i) = O(\tau_j)$ *still holds in the new graph. Analogously we could replace* $\rho^{l_a}_{l_i} o \rho^{l_b}_{l_a}$ *with* $\rho'^{l_b}_{l_i}$.

For the second part of the proof, as only the arc (l_a, l_b) *is deleted from* $\preceq$, *adding* (l_a, l_j) *and* (l_i, l_b) *as indicated above, transitivity still holds.*

By the operator's preconditions, $l_b \neq All$ *and* $l_a \neq l_{inf}$ *unless levels parallel to* l_b *or* l_a *exist. Thus, the new dimension will never have two top or bottom levels. In case* $l_a = l_{inf}$ *or* $l_b = All$, Unrelate *becomes a* Relate *operation, which correctness we have already proved.*

Delete Level *Here, there is only one case to study. We claim that* $O(\tau_1) = O(\tau_2)$ *still holds when we replace* $\rho^{l_j}_{l_i}$ *by* $\rho'^{l_j}_{l_i}$. *Let us suppose that* τ_1 *and* τ_2 *are two paths of the form* $O(\tau_1) = \rho^{l_s}_{l_1} o \ldots \rho^{l}_{l_i} o\ \rho^{l_j}_{l} o \ldots \rho^{l_m}_{l_t}$, *and* $O(\tau_2) = \rho^{l_r}_{l_1} o \ldots \rho^{l}_{l_k} o\ \rho^{l_n}_{l} o \ldots \rho^{l_m}_{l_u}$ *(it may occur that* $l_i = l_k$ *and* $l_j = l_n$*), where* l *is the deleted level. As both paths verify* $O(\tau_1) = O(\tau_2)$, *we could replace the rollup functions including* l *according to the operators definition, by the functions in* ρ'. *Thus,* $O(\tau_1) = \rho^{l_s}_{l_1} o \ldots \rho^{l_j}_{l_i} o \ldots \rho^{l_m}_{l_t}$, $= O(\tau_2) = \rho^{l_r}_{l_1} o \ldots \rho^{l_n}_{l_k} o \ldots \rho^{l_m}_{l_u}$.

Now we show that $\preceq'$ is still a partial order. By definition, the operator deletes the pairs of the form (l_i, l) and (l, l_j). Thus, $\preceq'$ will still be reflexive(every pair of the form (l_i, l_i) will remain). As all non-redundant arcs such that $l_1 \preceq l$ and $l \preceq l_2$ will be added, no cycles are induced. Thus, $\preceq'$ will be a partial order.

It is easy to show that G' will have a unique top and bottom. This follows from the operator preconditions, as l can neither be All *nor be l_{inf}(if l_{inf} only rolls up to one level).*

Add Instance *The schema conditions are preserved, as no schema change occurs in this operator. The inserted element could only be the initial element in any path it belongs to, because of the operator's preconditions (recall that elements are inserted "from bottom to top"). Thus, for every level l_a in a path of the form $\tau =< l_1, \ldots l_a, l_j \ldots l_m >$, when we add i_a to the instance set of l_a we also add rollup function instances of the form $(i_a, i_j), i_j \in instset(l_j)$, and these values must verify consistency. Then, for every pair of paths τ_1 and τ_2, $O(\tau_1) = \rho_{l_1}^{l_s} o \ldots \rho_{l_a}^{l_i} o \ldots \rho_{l_t}^{l_m} = O(\tau_2) = \rho_{l_1}^{l_t} o \ldots \rho_{l_a}^{l_j} o \ldots \rho_{l_k}^{l_m}$ in ρ', because i_a cannot affect this property.*

Delete Instance *Again, the schema conditions are preserved, as no schema change takes place here.*

At the instance level, as deletions occur from "top to bottom", i_a, the element to be deleted element could only be the first element in a path. It is obvious that consistency cannot be violated by such a deletion.

Theorem 2 *Given an initial dimension d, and a target dimension d', satisfying definitions 4 and 5, there is always a sequence of the operators 1 to 6 allowing to build d' starting from d.*

Proof Theorem 2 *We will proceed by induction on the number of nodes. We will prove that given a dimension d with N levels, we can build any dimension d' with $N+1$, $N-1$ or N levels applying operators 1 to 6. If this is possible, any dimension can be built with these operators.*

Let us consider the minimal possible dimension, this is, a dimension with levels l_{inf} and All. From this dimension, we can built only two possible ones, with $N = 3$. No new dimension with $N = 2$ could be built (we do not consider renaming the nodes, which is trivial). Figure 2.6 shows these possible dimensions. Dimension in Figure 2.6(b) is reached applying Generalize$(d, l_{inf}, l_n, f_{l_{inf}}^{l_n})$, *where $f_{l_{inf}}^{l_n}$ is any valid rollup function. Dimension in Figure 2.6(c) is built applying the following sequence:* Generalize$(d, l_{inf}, l_n, f_{l_{inf}}^{l_n})$, DelLevel$(d, l_{inf})$, Generalize$(d, l_n, l_{inf}, f_{l_n}^{l_{inf}})$.

Suppose now a dimension with N levels. Any dimension with $N+1$ nodes can be built in the following way: for any level l, apply Generalize$(d, l, l_n, f_l^{l_n})$ *in order to get a dimension with $N+1$ nodes. This new level can be related (if there is a consistency function) with any other existing level, because, by construction, the new level is independent from all the other ones. The only exception happens to be the case in which the inserted node becomes the new bottom level. This is called an* specialization *(Subsection 2.3.3). The following steps show the procedure: (a) delete all the nodes but l_{inf}. This is achieved applying exhaustively operator* DelLevel(d, l) *until a dimension with levels All and l_{inf} is reached; (b) proceed like in the minimal case explained above in order to insert l_n below l_{inf}; (c) rebuild the original dimension(by hypothesis, d was build with the operators, thus, it can be rebuilt). Step (a) above also shows that any valid dimension with $N-1$ nodes could be reached starting from d, just by deleting one level. Finally, given a dimension d with N levels, we can reach any other dimension d' with the same number of levels, applying the Relate and Unrelate operators, which allows to add or delete edges if preconditions are met.*

At the instance level the proof follows straightforwardly from the operator's definition. Given a dimension instance d, any possible valid instance could be built. Actually, for every element $i \in instset(l_{inf})$ we can apply $DelInstance(d, l_{inf}, i)$ in order to delete all the elements in this level. Proceeding in the same fashion we could build a dimension such that for every level l in $L \setminus \{All\}$, $instset(l) = \phi$. From this empty dimension instance, we could exhaustively apply the AddInstance operator, starting from every level l such that $l \preceq All$, and populate the dimension again.

Theorem 3 *Operators 1 to 6 are minimal, meaning that no subset of them can define all possible dimensions satisfying Definitions 4 and 5.*

Proof Theorem 3 *Let us analyze the structural operators one by one. We will prove that if one of them is left out, there is at least one dimension which could not be built. Let us denote by $\mathcal{O}_p$ the set of operators 1 to 6 defined above.*

Generalize. *It is trivial to show that the set defined by $\{\mathcal{O}_p \setminus$ Generalize$\}$ cannot define a dimension, because* Generalize *is the only operator allowing to insert a new node in the Directed Acyclic Graph (DAG) which represents a dimension.*

Relate and Unrelate. *In an analogous way, it can be proved that with the set $\{\mathcal{O}_p \setminus$ Relate$\}$ we cannot build every possible dimension d' starting from a dimension d. Given a dimension d with*

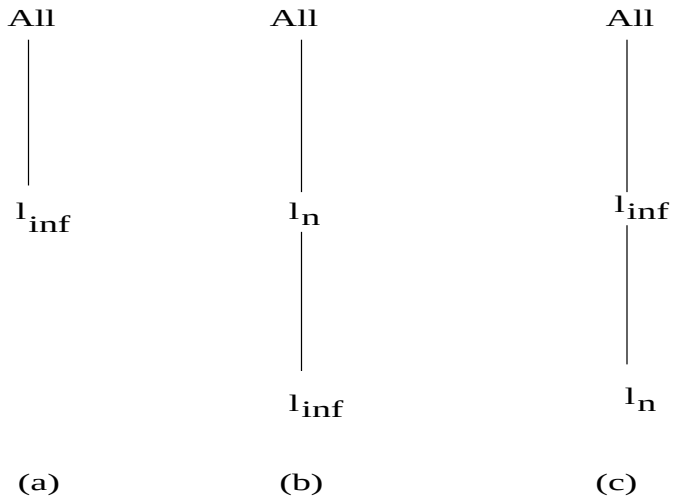

Figure 2.6: All possible three-level dimensions.

two parallel levels l_a and l_b such that $l_{inf} \preceq^* l_a$ and $l_{inf} \preceq^* l_c, l_c \preceq^* l_b$, there is not possible to add the edge l_a, l_b with the set $\{\mathcal{O}_p \setminus$ Relate$\}$. In this case, the only way to this would be inserting a level l_a below l_b, which would imply the deletion of all the dimension, except for the path $< l_b, \ldots All >$. However, doing this prevents inserting l_c below l_b. The proof for $Unrelate$ proceeds analogously.

Delete Level. *For this operator the proof is trivial, given that this is the only operator allowing to delete a node in the graph. The same occurs with* Add Instance *and* Del Instance.

2.3.3 Complex Structural Update Operators

The *Grain* of a multidimensional database is the finest granularity at which factual data is stored in the database. For instance, in Example 3, the grain of the database is defined by the set of levels $\{itemId, storeId, day\}$. Thus, the set $GBottom_D$ defines the database grain.

Refining the granularity of a dimension would be a usual situation in a data warehousing environment like the one we are proposing in the present work. For instance, in the example above, we may want to record sales by hour instead of by day. Inserting a new bottom level in a dimension can imply, in the worst case, deleting and rebuilding the whole dimension. Suppose a dimension with just two levels, *All* and l_{inf}. In order to insert a new level l_n below l_{inf}, we must generalize l_{inf} to l_n, then delete l_{inf}, and finally generalize again l_n to l_{inf}. It is clear, then, that an operator capable of performing this task in a straightforward way is needed. Thus, we define the *Specialize* operator.

The *Specialize* operator adds a new level l_n to a dimension. Level l_n will roll up to the lowest

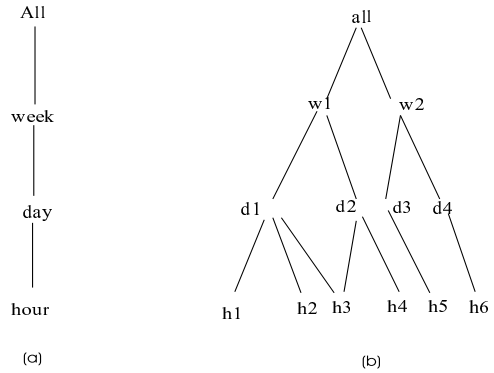

Figure 2.7: (a) Schema after Specialization (b) Instance after Specialization.

level of L, l_{inf}, becoming the lowest level of the dimension. A function must be defined for this rollup.

Operator 7 (Specialize) *Given a dimension* $d = ((dname, L, \preceq), \rho)$*, a level* $l_n \notin L$*, and a function* $f^{l}_{l_n} : dom(l_n) \rightarrow instset(l_{inf})$*,* Specialize$(d, l_n, f^{l_{inf}}_{l_n})$ *is a new dimension of the form* $((dname, L \cup \{l_n\}, \preceq \cup\{(l_n, l_{inf})\}), \rho')$*, where* ρ' *is the set containing the rollup functions* $\rho'^{l_j}_{l_i}$*, such that:*

- $\rho'^{l_{inf}}_{l_n} = f^{l_{inf}}_{l_n}$
- $\rho'^{l_j}_{l_i} = \rho^{l_j}_{l_i}$ *for all other levels* l_i, l_j.

Notice that $\rho'^{l_{inf}}_{l_n}$ *can be the empty set, according to Definition 5*

Example 10 *Specializing the* Time *dimension of example 3 will result in the dimension depicted in Figure 2.7. Note that the only level apt for specialization is* day*, and the operation is defined as* Specialize(Time, hour, f^{hour}_{day}).

In Section 2.5 we will define a set of complex instance update operators.

2.4 Maintenance

When an update to a dimension occurs, and a data cube has been materialized in the data warehouse, the cube views which conform this data cube must be updated in order to reflect the updates over the dimension. We would like to incrementally maintain the data cube. In this section we

address this problem, treating separately structural and instance updates. We will assume a relational storage of the multidimensional database(ROLAP). The mapping from the multidimensional to the relational model is defined as follows.

Fact tables are represented as relations such that there is an attribute for every level in the level group in the fact table, and an attribute for each measure (Definition 7). The relational representation of a dimension d is 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. The components of S_d and T_d are detailed below.

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

- $rname$ is $dname$;
- 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 $l_a \preceq l_b$.

Instance 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 $instSet(l)$ not reached by any other element below them in the dimension instance. Formally: $Leaves(l) = instSet(l) \setminus (ran(\rho^l_{l_1}) \cup \ldots ran(\rho^l_{l_n}))$, where $l_1, \ldots, l_n$ are the levels directly below l in the hierarchy.
- For every level l, and for every element $e \in Leaves(l)$, there is a tuple t in T_d defined as follows:

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

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

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

In order to specify the maintenance algorithms, we will use the relational algebra with bag semantics, extended with the *generalized projection operator* to express aggregation [GHQ95b].

The generalized projection operator will be denoted as Π_A, where A is a set of attributes which can include aggregate functions. Then, computing $CubeView(D,f_{base},GB)$ is equivalent to computing the relation:

$\Pi_{GB\setminus\{All\},Ag(m)} f_{base} \bowtie d_1 \ldots \bowtie d_n$,

where $d_1, \ldots, d_n$ are the dimensions corresponding to the levels that are in $GB \setminus \{All \cup GBottom_D\}$.

The algorithms presented in this section were implemented. The details of the implementation are given in Chapter 3. Experimental results are discussed in Chapter 4.

Notation When using the generalized projection, we will use GB instead of $GB \setminus \{All\}$. In this way, the cube view of example 3 can be expressed as:

$\Pi_{itemId,storeId,week,Sum(sales)} Daily\_Sales \bowtie Time$

Also, in what follows, although it is not a basic operator, we will include the *Specialize* operator in our discussion.

2.4.1 Structural Updates

When a dimension level is generalized, new views are generated as a consequence of the level being added. Thus, these new views must be computed from scratch. When a bottom level of some dimension is deleted or specialized, the base fact table must be updated in order to reflect these changes. To address this problem, we propose the following *data cube adaptation*:

- if a $DelLevel$ is specified over the bottom level of a dimension, recompute f_{base} as the cube view of the old base fact table grouped by the bottom level group of the new dimension set D'.

- If the update is $Specialize$, f_{base} will not longer be a base fact table of D', and the new base fact table over D' of which f_{base} is a cube view is not uniquely determined. We will not address the problem of finding an appropriate new base fact table, but it is easy to prove that one always exists. The determination of a new base fact table is related to the *reconstruction problem* [FJS97], which consists on the estimation and computation of a fact table based on an aggregate view of it.

The following algorithm performs incremental maintenance when a structural update occurs.

Algorithm 1

Input: *A data cube* dc, *and a structural dimension update* u *over a dimension* d. *We assume, w.l.o.g. a fully materialized data cube.*

Output: *A data cube incrementally maintained reflecting* u.

1. *if* u = Relate, *or* u = UnRelate, *no maintenance is required. We call these updates* irrelevant *with respect to data cube maintenance.*

2. *If* u=Generalize, *the new views which are generated (due to the presence of a new level) must be computed from scratch.*

3. *If the update is* $u = DelLevel(d, l)$:

   - *if* $l \in GBottom_D$, *compute a new base fact table as* $\Pi_{GBottom_{D'}, Ag(m)} f_{base} \bowtie d$ *(where* D' *is the new set of bottom levels);*
   - *drop every table* f *in* dc *such that* l *belongs to the schema of* f.

4. *If* $u = Specialize(d, l_0, \rho_{l_{inf}}^{l_0})$, *compute a fact table* f'_{base} *such that* $f_{base} = \Pi_{GBottom_D, Ag(m)} f'_{base} \bowtie d'$, *and drop* f_{base}.

In Section 4.4 we show the results of applying Algorithm 1 to a case study.

2.4.2 Instance Updates

An instance update in a relational OLAP implementation reduces to the insertion and/or deletion of some tuples in the dimension tables. We could therefore apply existing incremental maintenance algorithms for materialized relational views with aggregates [GMS93, Qua96, MQM97]. However, we will see that we can do better by exploiting the special form of these updates. In particular, we will show how to improve on the summary-delta algorithm introduced by Mumick et al. [MQM97].

In general, as we commented on Chapter 1, incremental maintenance involves:

1. computing the set of changes, sometimes called the delta table(*propagation phase*);

2. applying the changes represented in the delta table to the materialized view(*refresh phase*).

In the case of views with aggregations, this approach applies only if the aggregate functions are *self-maintainable* [MQM97]. Again, we will assume in what follows that the aggregate operator

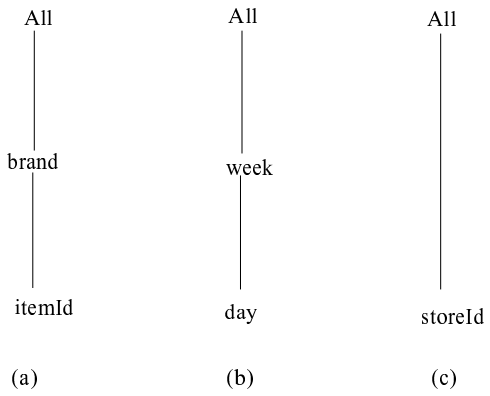

Figure 2.8: Dimension schemas for Example 11

Ag is *SUM*, and that we extend the tables and the cube views storing the *count* value required to make *SUM* self-maintainable with respect to insertions and deletions. The implementation to be described in Chapter 3 supports the five classic SQL aggregate functions.

In the next subsections we show in detail how that we can improve the summary-delta algorithm if we take advantage of the particular features of the kinds of instance updates proposed in this chapter.

In what follows, we will use a reduced version of the data cube of Example 3 in order to make it easy to understand the proposal.

Example 11 *To show the ideas depicted in this section, we will be considering the set of dimensions* D={Product, Store, Time, Sales}, *with the schemas specified in figure 2.8. The rollup functions are the following (the rollups to* All *are omitted, where possible):* $\rho^{brand}_{itemId} = \{i_1 \mapsto b_1, i_2 \mapsto b_2, i_3 \mapsto b_2, i_4 \mapsto b_3\}$; $\rho^{All}_{storeId} = \{s_1 \mapsto all, s_2 \mapsto all\}$; $\rho^{week}_{day} = \{d_1 \mapsto w_1, d_2 \mapsto w_1, d_3 \mapsto w_2\}$. *We will use the base fact table of Example 3, called* DailySales, *so we will not repeat it here. The data cube is defined as* dc=DataCube(D,f_{base},GBSET), *and each cube view will be denoted as "*Sales*" appended with the initials of the levels in the group level defining it (e.g.* Sales_ASD *represents the cube view such that* $GB = \{All, storeId, day\}$).

Maintaining Cube Views Independently

Under instance updates in a dimension, we can proceed as in the summary-delta algorithm, i.e., separately compute a maintenance expression for each Cube View with the rules presented by Quass [Qua96]. The maintenance expression produces the change to each cube view, which is then

Consider a cube view $f = CubeView(D, f_{base}, GB)$, where $GB = \{l_1, \ldots l_n\}$, and a fact table Δf with the same schema of f; $Refresh(f, \Delta f)$, computes f' as follows:
For every tuple $\delta t \in \Delta f$:

- Let $t = \Pi_{GB}\sigma_{l_1=\delta t(l_1),\ldots,l_n=\delta t(l_n)} f$
- If t is not found, insert tuple δt into f
- Else
  - If $t(count) + \delta t(count) = 0$ delete t from f
  - Else $t(count) += t(count)$, $t(m) = Ag(\{t(m), \delta t(m)\})$, where m is the measure of the fact table.

Figure 2.9: Refresh operator adapted from the summary-delta algorithm.

applied to the cube view itself using a *refresh algorithm*. We define $Refresh(f, \Delta f)$ to be the result of applying the refresh algorithm to a cube view f, given a set of changes Δf, yielding a new cube view $f' = CubeView(D', f_{base}, GB)$, where D' is the set of dimensions after the update.

The approach in the summary-delta algorithm is focused on a deferred maintenance under a large delta change, which is not the case for the operators defined in this chapter. Thus, we can derive efficient maintenance expressions in order to compute changes to the cube views. First, let us define the meaning of the change to a cube view computed with the summary-delta algorithm.

Definition 12 (Cube View Change) *Given a cube view $f = CubeView(D, f_{base}, GB)$, and an instance update* u*, such that D' is the updated dimension set, the* cube view change *of f with respect to* u*, denoted Δf, is the fact table which satisfies: $f' = Refresh(f, \Delta f)$, where $f' = CubeView(D', f_{base}, GB)$. The refresh operator is shown in Figure 2.9.*

Lemma 1 (Irrelevant Updates) *Given a dimension d, a dimension update $AddInstance(d, l, i, P)$ or $DelInstance(d, l, i)$ is irrelevant(i.e. the dimension updates do not have an effect over any cube view) if l is not the bottom level of d, or $\sigma_{l=i} f_{base} = \phi$.*

Proof Lemma 1 *Consider an update $DelInstance(d, l, i)$. According to the way in which elements are deleted from d, if $l \neq l_{inf}$ no fact could be associated with i, because by the preconditions of the operator there could be no element j such that $\rho^{*l}_{l_{inf}}(j) = i$. If $\sigma_{l=i} f_{base} = \phi$, then i does not affect*

Given a instance update u, and a cube view $f = CubeView(D, f_{base}, GB)$

1 If $u = Delinstance(d, l, i)$, and there exists another view $f_1 = (D, f_{base}, GB_1)$, where $GB_1 \setminus GB = \{l\}$, then:

$\Delta f = \Pi_{GB\setminus\{l\},l'=\delta^- d(l'),m=-m}\sigma_{l=i} f_1$, where $l' = GB \setminus GB_1$

2 If $u = Delinstance(d, l, i)$ and $GB_1 \setminus GB \neq \{l\}$ then $\Delta f = \Pi_{GB\setminus\{l'\},l'=\delta^- d(l'),m=-Ag(m)}\sigma_{l=i} f_{base} \bowtie d_1 \bowtie \ldots \bowtie d_n$, where l' is the level in GB that belongs to d, and $d_1 \ldots d_n$ are the dimensions such that the levels in $GB \setminus \{d\} \setminus GBottom_D$, belong to.

3 If $u = Addinstance(d, l, i, P)$ then

$\Delta f = \Pi_{GB\setminus\{l'\},l'=\delta^+ d(l),m=Ag(m)}\sigma_{l=i} f_{base} \bowtie d_1 \bowtie \ldots \bowtie d_n$, where l' is the level in GB that belongs to d, and $d_1 \ldots d_n$ are the dimensions such that the levels in $GB \setminus \{d\} \setminus GBottom_D$, belong to.

Figure 2.10: Maintenance expressions for instance updates.

any element in the fact table. Thus, element $i \in instset(l)$ does not contribute to any view derived from f_{base}. The proof proceeds analogously for $AddInstance(d, l, i, P)$.

If an update is relevant, the delta changes that have an effect over the cube views are: $\Delta^+ d = \sigma_{l=i} d'$, in case of an *AddInstance*, and $\Delta^- d = \sigma_{l=i} d$, in case of a *DelInstance*, both containing a single tuple, which we will denote $\delta^+ d$ and $\delta^- d$ respectively.

Figure 2.10 shows a set of rules for independent view maintenance under instance updates in a dimension. Notice that if Rule 1 holds, no aggregation is required.

Example 12 *Consider the data cube of Example 11, and an update* DelInstance(Product,itemId,i_2) *in the* Product *dimension, represented by $\Delta^- Product$. Also consider the fact table DailySales of Example 3. The maintenance expression for the cube view Sales_BSW=$CubeView(D, f_{base}, \{brand, store, week\})$, would be:*

$\Delta Sales\_BSW = \Pi_{brand=b_2,storeId,week,sales=-sales}\sigma_{brand=b2} Daily\_Sales \bowtie \Delta^- Product \bowtie Time.$

The new cube view, $CubeView(D', Daily\_Sales, \{brand, store, week\})$, is computed, in the refreshing stage, from Sales_BSW and the delta change of f, using the Refresh algorithm. *Thus, $Sales\_BSW' = Refresh(Sales\_BSW, \Delta Sales\_BSW)$. This is done separately for every cube view in dc.*

Finally, the incremental maintenance algorithm can be stated as follows:

Algorithm 2

Input: *A data cube* dc, *and a dimension update* u *over a dimension* d. *We assume, w.l.o.g. a fully materialized data cube.*

Output: *The updated data cube.*

For each cube view cv *in* dc

If u *is irrelevant* skip

Else

compute the changes Δdc *using the rules in figure 2.10;*

Refresh(dc,Δdc)

Example 13 *Consider the data cube of Example 11 and the update* DelInstance(Product,ItemId,i_2). *Using rule 1, we can compute the delta change associated to a cube view DeltaSales_BSW, from the already materialized cube view Sales_ISW (see Example 3), with the following expression, avoiding any aggregate computation. The resulting table is also shown.*

$\Delta Sales\_BSW = \Pi_{brand=b_2,storeId,day,sales=-sales}\sigma_{itemId=i_2}\ Sales\_ISW.$

brand	storeId	week	sales
b_2	s_1	w_1	*-20*
b_2	s_2	w_1	*-60*

After this, the new cube view could be computed using the *Refresh* operator.

Maintenance Using the View Derived Lattice

In the former section we showed how to compute a cube view change from cube views which were already materialized. We would like to compute cube view changes from other *cube view changes,* leading to a more efficient implementation. For instance, in Example 13, we would compute $\Delta Sales\_BSW$ from $\Delta Sales\_ISW$.

Below, we present an algorithm for incremental view maintenance in the presence of instance updates, by adapting the view lattice introduced by Harinarayan et al [HRU96], to our specific needs.

The changes to the base fact tables, under dimension instance updates, are given by:

- $\Delta f_{base} = \Pi_{GBottom_D,m=-m}\sigma_{l=i} f_{base}$, if the update is $DelInstance(d,l,i)$;

- $\Delta f_{base} = \sigma_{l=i} f_{base}$ if the update is $AddInstance(d, l, i, P)$.

Then the expression $\Delta dc = DataCube(D \setminus \{d\} \cup \Delta d, \Delta f_{base}, GBSET)$, where $\Delta d = \delta^- d = \sigma_{l=i} d$ for $DelInstance$, and $\Delta d = \delta^+ d = \sigma_{l=i} d'$ for an $AddInstance$, defines a *delta cube.*

The delta cube defined in this way contains the cube view changes to the data cube $dc = (D, f_{base}, GBSET)$. Thus, we can apply the summary-delta algorithm to compute the delta cube Δdc, taking advantage of the fact that Δd contains only one tuple.

Again, we will assume that $GBSET = GB_D$, i.e., the data cube is fully materialized. The algorithm that we propose can be extended to a partially materialized data cube.

We define the *view lattice* as follows:

- there is a node, denoted $N(GB)$, for each level group GB in GB_D.
- given two nodes $N(GB)_1$ and $N(GB)_2$, for each 1-1 function co between GB_1 and GB_2 such that for each level l in GB_1, $l \preceq co(l)$, there is an edge from $N(GB_1)$ to $N(GB_2)$, and this edge is labeled by the dimensions $d_1, \ldots, d_n$ of each level l in GB_1 such that $l \neq co(l)$. We denote this edge $N(GB_1) \preceq_{d_1,\ldots,d_n} N(GB_2)$.
- each edge $N(GB_1) \preceq_{d_1 \ldots d_n} N(GB_2)$ is associated with a bag algebra expression which computes the cube view change Δf_2 associated to GB_2 from the cube view change Δf_1 associated to GB_1: $\Delta f_2 = \Pi_{GB_2, Ag(m)} \Delta f_1 \bowtie d_1 \bowtie \ldots \bowtie d_n$.

Figure 2.11 shows a view lattice for a base fact table with $LSET = \{itemId, storeId, day\}$. We have labeled each node with the initials of the levels in each level group GB. We have omitted the edges' labels, for the sake of the figure's clarity.

We now present a set of rules that, given an update to level l in dimension d, modify the algebra expressions associated with the edges of the view lattice. We only consider "direct" edges, that is, those edges between sets GB_1 and GB_2 that differ only on one level. It is worth noting that the optimization depicted in rule C below is also present in the work of Mumick et al.([MQM97]), and avoids repeating joins along the lattice paths, by means of performing the join with dimension d the first time it is required and preserving the levels of the dimension for computing the cube view changes upward in the lattice.

Given a dimension update over a dimension d and a level l, the rules which modify the lattice are:

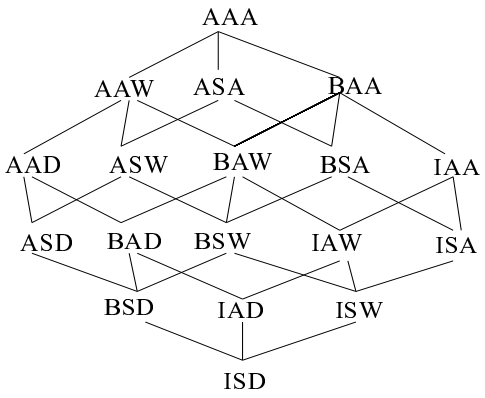

Figure 2.11: A view lattice for the running example.

RULE A For the direct edges $N(GB_1) \preceq_d N(GB_2)$ such that $l \in GB_1$, $GB_1 \setminus GB_2 = \{l\}$,

$\Delta f_2 = \Pi_{GB_2 \setminus \{l'\}, l_1 = \delta d(l), m} \Delta f_1$;

$\Delta f_2{}' = \Pi_{GB_2 \setminus \{l'\}, l_1 = \delta d(l_1), \ldots l_n = \delta d(l_n), m} \Delta f_1$,

where δd is $\delta^+ d$, if the update is $AddInstance(d, l, i, P)$ or $\delta^- d$ if it is $DelInstance(d, l, i)$, and m is considered as $-m$ in case of a $DelInstance$; l_i are the levels in d, and $l' = GB_2 \setminus GB_1$.

$\Delta f_2{}'$ is computed in order to propagate the view change upward in the lattice without performing any join (see explanation below).

RULE B For the direct edges $N(GB_1) \preceq_d N(GB_2)$ s.t. $l \notin GB_1$ and $GB_1 \setminus GB_2 = \{l'\}, l' \in d$, $\Delta f_2 = \Pi_{GB_2} \Delta f_1{}'$.

RULE C The remaining direct edges, of the form $N(GB_1) \preceq_{d_i} N(GB_2)$(for instance, edges linking views that differ in levels belonging to dimensions other than d, the dimension being updated), are modified in the following way :

For each level $l_i \in d$, consider the nodes $N(GB)$ such that $l_i \in GB$, and the edges between these nodes. These edges form a sub-lattice of the view lattice (see Figure 2.12). Let τ be a path including the bottom level of this sub-lattice and two nodes $N(GB_1)$ and $N(GB_2)$ such that $N(GB_1) \preceq_{d_i} N(GB_2)$. If the join with d_i was already performed in τ(i.e., $\tau =< \ldots, N(GB_i) \preceq_{d_i} N(GB_j), \ldots, N(GB_1) \preceq_{d_i} N(GB_2) \ldots >$), change the expression corresponding to the edge to:

$\Delta f_2 = \Pi_{GB_2, Ag(m)} \Delta f_1{}'$, where $\Delta f_1{}'$ has the same meaning as in Rules A and B.

If not (i.e., this is the first occurrence of d_i in the path), then change the derived expression to

$\Delta f_2 = \Pi_{GB_2, Ag(m)} \Delta f_1 \bowtie d_i$, and also generate $\Delta f_2{}'$ with all the attributes, in order to propagate the join upward in the lattice.

In Rule A all the attributes of the tuple added or deleted (δd) associated to the dimension change are stored in the cube view change($\Delta f_2{}'$). Thus, further derivations using rule B just

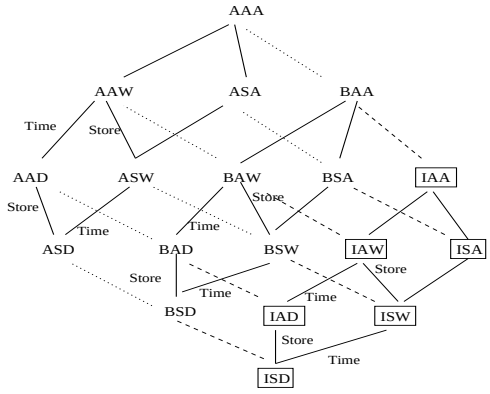

Figure 2.12: An optimized view lattice for the running example.

require copying the previous delta view. The view is updated using Δf_2, without computing any aggregation or join.

In Rule B, the delta view is obtained as a projection of the previous view change, over the view attributes. Observe the use of $\Delta f_1{}'$, i.e., the cube change propagated from views below f_1 in the lattice.

Example 14 *An optimized lattice for our running example is depicted in figure 2.12. The dashed lines represent the derived expressions changed using rule A; the dotted lines represent the derived expressions changed using rule B; the solid lines represent the derived expressions changed with rule C. Here, the edges are labeled with a dimension name when the join with the dimension must be included in the expression.*

Consider for example an instance update $DelInstance(Product, itemId, i_2)$. *For our running data cube, the delta view* $\Delta Sales\_BSW$ *would be computed using Rule A. The delta view at the bottom of the lattice,* $\Delta Sales\_ISD$, *will be:*

itemId	storeId	day	sales
i_2	s_1	d_1	*-20*
i_2	s_2	d_1	*-20*
i_2	s_2	d_2	*-40*

Thus, applying Rule A, $\Delta Sales\_BSD$ *would only require to replace every* i_2 *in the first column, by* b_2. *Recall that we still store the tuple* $< i_2, b_2, all >$, *thus, we can apply Rule B for computing the view* $\Delta Sales\_ASD$, *from* $\Delta Sales\_BSD$. *In this case,* $\Delta Sales\_ISD'$ *will be:*

brand	All	storeId	day	sales
b_2	*all*	s_1	d_1	*-20*
b_2	*all*	s_2	d_1	*-20*
b_2	*all*	s_2	d_2	*-40*

It is worth noting that computing the view $\Delta Sales\_ASW$ in Example 14 does not require a join with the $Time$ dimension, as it is in a path including $\Delta Sales\_BSD \preceq_{Time} \Delta Sales\_BSW$, and we can choose the delta view $\Delta Sales\_BSW$ as a predecessor. The only views in Example 14 which require a join, are the ones inside a box in Figure 2.12.

It is now straightforward to devise an algorithm which generates a plan leading to reduce the number of joins and aggregate computations while performing view maintenance due to instance updates.

Algorithm 3

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

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

- *For each node $N(GB)$ such that $l \in GB$:*
  - *if $u = DelInstance(d, l, i)$*
    - *For each node $N(GB)$ such that $l \in GB$, derive its view change using the expressions of Figure 2.10 and Rule C;*
  - *if $u = AddInstance(d, l, i, P)$*
    - *choose a node GB', immediately below GB, using one of the well-known methods for computing aggregates (v.g. [AGS+96]). As a default, the predecessor with the least estimated size could be used;*
- *For each node $N(GB)$ such that $l \notin GB$, choose as predecessor the node $N(GB')$, immediately below (i.e., connected by a direct edge), which differs with GB in only one level in d, giving priority to changes which can be computed using Rules A or B.*

Note that Algorithm 3 only in the case of an $AddInstance$, and for the nodes $N(GB)$ such that l is in GB, computes the cube view using derived expressions containing aggregations and joins.

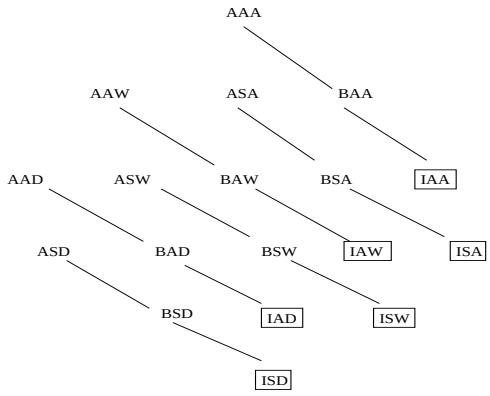

Figure 2.13: The plan generated for update $DelInstance(Product, Item, i_2)$.

In Chapter 3 we compare the performance of this algorithm for a DelInstance operation and the SUM aggregate function, against the plain summary-delta algorithm.

Example 15 *Figure 2.13 shows the graph for the plan derived for the data cube of example 11, for a* $DelInstance(Product, ItemId, i_2)$ *update. The nodes inside a box represent the cube views such that* $l \in GB$*. The sub-lattice indicates which predecessors must be chosen. We can proceed analogously for the case of an AddInstance operation.*

2.5 Complex Instance Update Operators

In Section 2.3 we introduced a set of operators which allow updating dimensions in the multidimensional model presented in Section 2.2. Moreover, in Section 2.3.3 we introduced the *Specialize* operator, which can be defined in terms of the basic structural update operators. At the instance level, many common updates to dimensions would result in long sequences of primitive updates, as we will show below. Thus, a set of *complex instance update operators* is needed in order to capture such common sequences and encapsulate them in a single operation.

Although the set of complex operators we will introduce is not the only one we could devise, we will show through examples in this section, and through a case-study in Chapter 3, that they represent quite accurately many usual real-life situations.

The complex instance update operators we will define are:

- Reclassify
- Split

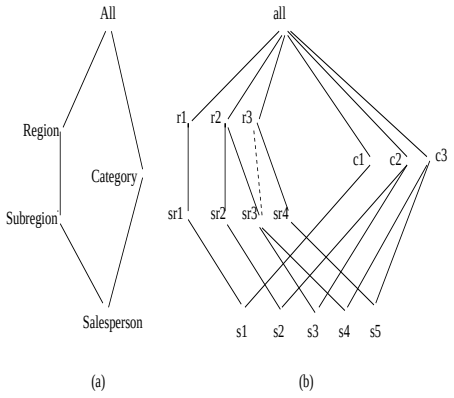

Figure 2.14: Reclassification

- Merge
- Update

2.5.1 Reclassify

Let us add a dimension *Salespersons* to our running example, with schema and instance depicted in Figure 2.14. Salespersons are grouped into *subregions* and *regions*. An orthogonal classification also groups salespersons into different *categories*. Initially, as Figure 2.14 shows, subregion sr_3 belongs to region r_3. At some time it is assigned region r_2. It is clear that this could be performed in a data warehousing environment like the one introduced in Section 2.2 as a transaction involving a series of $DelInstance$ and $AddInstance$ operations. Due to the preconditions of the primitive operators, six such operations would be necessary. Let us analyze how many graph operations this entails on the graph representing the dimension instance. $DelInstance$(d, salespersons, s_3), and $DelInstance$(d, salespersons, s_4) require two edge deletions each. $DelInstance$ (d, subregion, sr_3) takes an additional edge deletion. $AddInstance$ (d, subregion, sr_3,$\{r_2\}$) takes one edge addition, and finally $AddInstance$ (d, salespersons, s_3,$\{sr_3, c_2\}$) and $AddInstance$ (d, salespersons, s_4,$\{sr_3, c_3\}$) take two edge additions each. This gives a total of ten graph operations over the dimension *Salespersons*. On the other hand, it is clear that this could be done in just two steps: deleting the edge (sr_3, r_2), and adding the edge (sr_3, r_3). This operation is denoted $Reclassification$, and we will express it as $Reclassify(Salespersons, subregion, region, sr_3, r_3)$.

Operator 8 (Reclassify) *Given a dimension* $d=((dname, L, \preceq), \rho)$*, a pair of levels* l_a *and* l_b*, a pair of elements* $x_a \in instset(l_a)$ *and* $x_b \in instset(l_b)$; Reclassify(d, $\mathrm{l_a}$, $\mathrm{l_b}$, $\mathrm{x_a}$, $\mathrm{x_b}$) *is a new dimension*

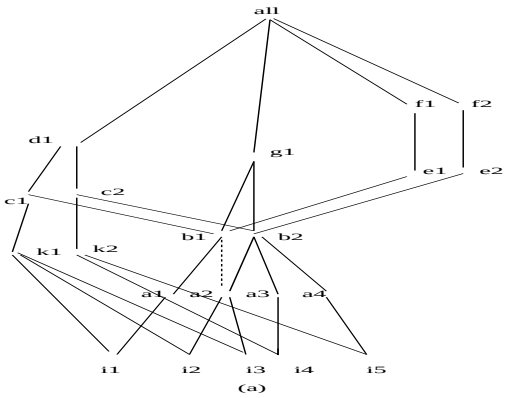

Figure 2.15: Reclassification

$((dname, L, \preceq), \rho')$ *s.t.:*

- $\rho'^{l_b}_{l_a} = \rho^{l_b}_{l_a} \setminus \{(x_a, x_j) | \rho^{l_b}_{l_a}(x_a) = x_j\} \cup \{(x_a, x_b)\}$;
- $\rho'^{l_j}_{l_i} = \rho^{l_j}_{l_i}$, *for all other levels* l_i, l_j.
- *The new dimension remains consistent.*

Reclassify is not defined for every possible dimension, as is shown in Figure 2.15. In this abstract dimension, if we reclassify a_2 from b_1 to b_2, consistency would be violated, because element i_3 in the bottom level could reach different elements in level C. However, note that as level B is the only one that rolls up to level E, the fact that b_1 and b_2 roll up to different elements in $instset(E)$ does not prevent reclassification. This observation leads to a general condition under which a reclassification may always be performed.

Definition 13 (Conflicting Levels) *Let us suppose we have a dimension d, two levels* l_a, l_b, *such that* $l_a \preceq l_b$, *and two elements* $x_a \in l_a, x_b \in l_b$, *defined as in operator 8. We say a level* $l_k \in d$ *s.t.* $l_b \preceq^* l_k$, *is* conflicting *w.r.t. reclassification, if there exists a level* l_i *s.t.* $l_i \preceq^* l_a$, *there is an alternative path between* l_i *and* l_k *not including* (l_a, l_b), *and* x_a *is reached by at least one element in* $instset(l_i)$. *A conflicting level is* minimal *if it is not reachable from any other conflicting level.*

Lemma 2 (Definiteness of the Reclassify Operator) $\mathrm{Reclassify(d, l_a, l_b, x_a, x_b)}$ *is defined if and only if:*

- *there are no conflicting levels in d*;
- *for every minimal conflicting level* l_k, $\rho^{*l_k}_{l_b}(x) = \rho^{*l_k}_{l_b}(x_b)$ *holds, where* $\rho^{l_b}_{l_a}(x_a) = x$.

Proof Lemma 2 . *For the "only if," suppose that l_k is a conflicting level, l_i is defined as in Definition 13, and there exists an element $e \in instset(l_i)$ s.t. e reaches x_a. Moreover, suppose that $\rho^{*l_k}_{l_b}(x_b) = e_1, \rho^{*l_k}_{l_b}(x) = e_2, e_1 \neq e_2$. In this case, it is clear that* Reclassify(d, l_a, l_b, x_a, x_b) *would lead to an inconsistent graph, because $\rho^{*l_k}_{l_i}(e) = e_1$, and $\rho^{*l_k}_{l_i}(e) = e_2$, through two different paths.*

For the "if," suppose that Reclassify(d, l_a, l_b, x_a, x_b) *is defined; thus, it must lead to a consistent dimension. There are two possible cases: (1) no conflicting levels exist, in which case, nothing else is required, and the reclassification only implies adjusting the rollup function $\rho^{l_b}_{l_a}(x_a)$ from x to x_b, or (2) l_k is a conflicting level w.r.t the proposed reclassification. In this case, let us suppose, again, that $e \in instset(l_i)$ exists, and $\rho^{*l_a}_{l_i}(e) = x_a$. As l_k is conflicting, if $\rho^{*l_k}_{l_i}(e) = e_1$ through a path not including (l_a, l_b) because, by hypothesis, reclassify is defined, this must hold through any path, including one passing through l_a, l_b or both. Thus, $\rho^{*l_k}_{l_b}(x) = \rho^{*l_k}_{l_b}(x_b)$ must hold.*

From Lemma 2 it follows that in a relational implementation, a reclassification is always more efficient than the equivalent sequence of primitive operators, because when the preconditions hold, updating the rollup function takes only one tuple insertion and one tuple deletion (see Chapter 3).

2.5.2 Split

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

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

- $\rho'^{l_a}_{l_i} = \rho^{l_a}_{l_i} \setminus \{(x_i, x_a) | \rho^{l_a}_{l_i}(x_i) = x_a \cup \{(x_i, x_{aj}) | x_{aj} \in E, x_i \in list_i \text{ s.t. } x_{aj}[l_i : list_i] \in P\}$;
- $\rho'^{l_i}_{l_a} = \rho^{l_i}_{l_a} \setminus \{(x_a, x_i) | \rho^{l_i}_{l_a}(x_a) = x_i\} \cup \{(x_{aj}, x_i) | x_{aj}[l_i : list_i] \in P, x_{aj} \in E \wedge \rho^{l_i}_{l_a}(x_a) = x_i\}$

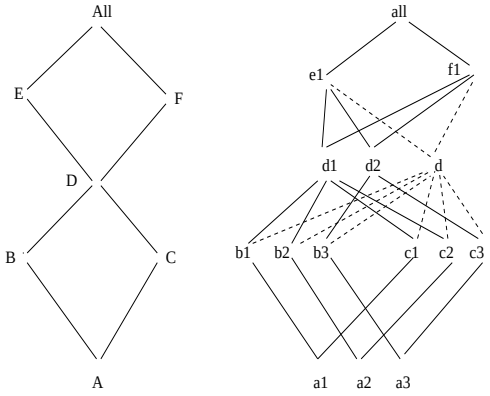

Figure 2.16: Split operator.

- $\rho'^{l_j}_{l_i} = \rho^{l_j}_{l_i}$, *for all other levels* l_i, l_j.
- *The new dimension must remain consistent.*

Example 16 *Figure 2.16 shows an abstract dimension. Suppose we want to split element* d *into two elements* d_1 *and* d_2 *in the domain of level* D. *The operation will be denoted as:* $\mathrm{Split}(dname, D, d, \{d_1, d_2\}, \{d_1 : (B : (b_1, b_2)), (C : (c_1, c_2)); d_2 : (B : (b_3)), (C : (c_3))\})$. *This expression assigns a set of elements in every level reaching directly level D(i.e. B and C), to each new element into which d splits (i.e.* d_1 *and* d_2*). Note that the user must assign the rollup functions corresponding to the new values* d_1 *and* d_2*, and this assignment must be s.t. the dimension remains consistent. In this example,* b_1, b_2, c_1 *and* c_2*, were assigned to* d_1.

Definition 14 (Conflicting Levels for Split) *A level* l_a *is* conflicting *with respect to* $\mathrm{Split}(d, l_a, x_a, E, P)$ *if there exist at least two levels* l_i *and* l_j *such that* $l_i \prec l_a$ *and* $l_j \prec l_a$.

Lemma 3 (Definiteness of Split) *Given a split operation* $\mathrm{Split}(\mathrm{d}, \mathrm{l_a}, \mathrm{x_a}, \mathrm{E}, \mathrm{P})$, *if* l_a *is not a conflicting level with respect to split, any list P defined as in Operator 9 will leave the dimension consistent after the operation.*

Proof Lemma 3 *It is trivial to see that if there is at most one level l such that* $l \prec l_a$, *any valid mapping defined by P will yield a consistent dimension. For every level l such that* $l_a \prec l$, *by definition, every element in E will roll up to the element in* $instset(l)$ *to which* x_a *rolled up before the split. Thus, no consistency violation could be possible "upward" in the lattice. If there is at most one level* l_d *such that* $l_d \prec l_a$ *(by hypothesis there are no conflicting levels), no consistency violation could occur "downward" in the lattice.*

The former property is relevant when implementing the Split operator, because it makes it possible to avoid the costly consistency checking. For instance, in Figure 2.16, the only conflicting level is D. If a Split over any other level occurs, no consistency checking must be performed.

2.5.3 Merge.

The *Merge* operator performs the inverse of *Split*, i.e., it merges two or more elements in a dimension level into a single one. For example, several airlines could become a single one as a result of a corporate fusion. Or, in our salespersons example, two subregions(e.g. north east and north east) could become a single one(e.g. north).

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

- $\rho'^{l_a}_{l_i} = \rho^{l_a}_{l_i} \setminus \{(x_i, x_j) | \rho^{l_a}_{l_i}(x_i) = x_j, x_j \in X\} \cup \{(x_i, x_N) | \rho^{l_a}_{l_i}(x_i) = x_j, x_j \in X\}$;
- $\rho'^{l_j}_{l_a} = \rho^{l_j}_{l_i} \setminus \{(x_i, x_j)\} \cup \{(x_N, x_j)\}, x_i \in X, \rho^{l_j}_{l_a}(x_i) = x_j, l_a \preceq l_j$;
- $\rho'^{l_j}_{l_i} = \rho^{l_j}_{l_i}$*, for all other levels* l_i, l_j.
- *The new dimension remains consistent, i.e., verifies all the properties stated in definition 5.*

Example 17 *Figure 2.17 shows an example of the above definition. The operation* $\mathrm{Merge(d, E, \{e_2, e_3\}, X_n)}$ *keeps the dimension in a consistent state.*

Note that, for a somewhat more general definition, we could have omitted the condition stating that every element in X must rollup to a single element in all the levels above l_a in the hierarchy. However, this condition makes *Merge* and *Split* symmetric, that is, $Merge(Split(d)) = d$. For instance, in Figure 2.16, if after the split we perform a merge of the form $Merge(dname, D, \{d_1, d_2\}, d)$, this operation would turn the dimension back to its original configuration.

Let us now give the intuition of why *Merge* may be more efficient than the equivalent sequence of $AddInstance$ and $DelInstance$ operations. Consider the following merge operation in our running example: $Merge(Salespersons, subregion, s_n, \{sr_2, sr_3\})$. This operation would take three edge

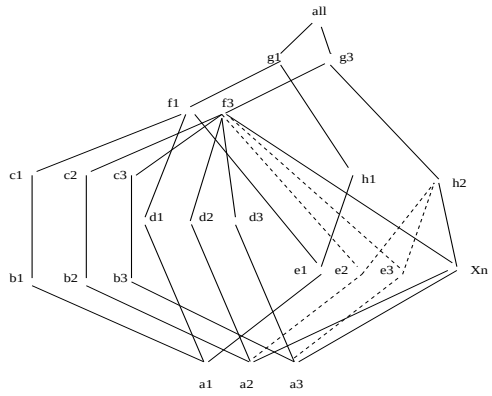

Figure 2.17: Merge operator

additions and five deletions, after checking that the preconditions for the operator hold. On the other hand, performing sequences of basic updates, it would take eight edge deletions over the graph, plus seven edge additions to obtain the same result. This pattern remains valid for most dimension schema and instances configurations.

2.5.4 Update.

Finally, we need an operator which only changes the value of an element, keeping the structure and rollup functions unchanged. For instance, suppose a brand name changes, say, for marketing reasons, but the Company owning the brand remains the same, as well as the set of products holding the brand. Again, all these changes could be reflected by a sequence of individual insertions and deletions, but simply renaming the brand would be a more efficient solution.

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

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

2.6 Summary

In this chapter we have introduced the concept of dimension updates in a multidimensional model. We have presented the model and given an algorithms for efficient data cube maintenance in the presence of these updates. We also introduced a set of basic and complex operators which update the structure and instances of a dimension. We claim that, although usually overlooked, allowing a dimension to evolve instead of rebuilding it from scratch when an update occurs, is a feature that adds value to a data warehouse implementation.

Chapter 3

Implementation of Dimension Updates

In this chapter we present a relational implementation of the model introduced in Chapter 2. The system was built following the OLEDB for OLAP standard. The syntax for the update operators is an extension of MDX, the emerging standard language proposed by Microsoft for multidimensional databases [Mic98]. This choice was made taking into account the wide set of client tools which can interact with OLEDB for OLAP providers, and the growing number of opinions considering MDX as a potential standard for access and manipulation of multidimensional data [TBG+99]. Thus, we implemented an OLEDB for OLAP provider which we called TSOLAP, and a data visualization tool, TSShow, allowing to graphically display the structure and instances of a set of dimensions.

This chapter is organized as follows: in Section 3.1 we discuss the mapping from the multidimensional to the relational model. In Section 3.2 we report experimental results comparing how two relational representations of the multidimensional model perform with respect to dimension updates. Section 3.3 presents a short review of OLEDB for OLAP and MDX standards. In Section 3.4 we describe the implementation of TSOLAP. In Section 3.5 we describe how TSOLAP could be used by OLE DB for OLAP consumers, and introduce TSShow. We conclude in Section 3.6.

3.1 Mapping Dimensions to Relations

In this section we discuss different ways in which the dimensions described in Chapter 2 can be represented in the relational model. We study two possible implementations:

- dimensions stored as a single table(denormalized representation);
- dimensions stored as a set of normalized tables(normalized representation).

3.1.1 Denormalized Relational Representation

In Section 2.4 we defined the relational representation of a dimension d as a tuple $R_d = (S_d, T_d)$, where $S_d = (rname, A, \mathcal{F})$ is the schema of the relation and T_d is the set of tuples in the relation representing d. The set $\mathcal{F}$ contains a functional dependency $l_a \rightarrow l_b$ for each pair of levels $l_a, l_b \in L$ such that $l_a \preceq l_b$, and A contains an attribute l for each level $l \in L$. We call this representation *denormalized*. Let us describe the denormalized relational representation in more detail.

Definition 15 (Weak Dependency) *Given a relation schema R, and a functional dependency F, we define F as* weak *if for every instance $r \in R$, F holds only over tuples in r that do not include null values.*

The functional dependencies in $\mathcal{F}$ are weak, because the model allows dimensions where a level $l \neq l_{inf}$ could be such that $card(Leaves(l)) \geq 2$ (i.e., dimensions where at least two elements in $instSet(l)$ are not reached by any element in a level below l).

Example 18 *The denormalized representation of the dimension depicted in Figure 2.5(b) is the following:*

itemId	brand	company	category
$null$	b_1	co_1	$null$
i_2	b_2	co_1	c_1
i_3	b_3	co_2	c_2
i_4	b_3	co_2	c_2
i_5	b_3	co_2	c_2

The following functional dependencies hold: $\mathcal{F} = \{itemId \rightarrow brand, brand \rightarrow company, itemId \rightarrow category\}$. Inserting a new brand, e.g. b_5 for company co_1, will add a tuple $<null, b_5, co_1, null>$. Thus, a null value will be associated with two different brands $(card(Leaves(brand)) = 2)$. This shows that the functional dependencies above are weak.

Lemma 4 *Given a dimension $d = ((dname, L, \preceq), \rho)$, and its relational representation $R_d = ((rname, A, \mathcal{F}), T_d)$ built in the fashion described above, the following hold:*

- *The set $\mathcal{F}$ is a minimal cover of itself, and it is the only possible one.*
- *T_d represents the transitive closure ρ^* of the set of rollup functions ρ.*

Proof Lemma 4 . *For the first part of the lemma, we have that, by construction, no side of any dependency has more than one attribute, each one representing a dimension level. Moreover, as $\preceq$ does not include redundant edges, the mapping defined in Section 2.4 implies that no dependency can be redundant. These conditions define a minimal cover of a set of functional dependencies. If we now consider a pair of levels l_i and l_j in a dimension d, such that $l_i \preceq l_j$, this relation defines the dependency $l_i \rightarrow l_j$ (by Definition 3.1.1). We cannot delete any dependency and still have the same dimension being represented.*

For the second part of the lemma, let us suppose a dimension d including levels l, l_1, l_i, l_j, where $l \preceq^* l_1, l \preceq^* l_i, l \preceq^* l_j$, $l_i \preceq^* l_1$, $l_j \preceq^* l_1$, $l_i \not\preceq^* l_j$, and $l_j \not\preceq^* l_i$. According to the mapping defined above, these levels are represented in the relational model as attributes of a relational schema S_d, named A_l, A_{l_1}, A_{l_i}, and A_{l_j}. Let us suppose now that $\rho_l^{*l_i}(a_1) = a_2, \rho_l^{*l_j}(a_1) = a_3, \rho_{l_j}^{*l_1}(a_3) = a_4, \rho_{l_i}^{*l_1}(a_2) = a_4$, satisfying the consistency condition. This will be represented in T_d by a tuple of the form $(a_1, .., a_2, .., a_3, ...a_4)$. This tuple includes all the possible paths from a_1 to a_4, which are $< a_1, a_2, a_4 >$ and $< a_1, a_3, a_4 >$. Furthermore, this holds for every tuple in T_d.

3.1.2 Normalized Relational Representation

In an environment in which dimension updates are supported, it is worth to compare the denormalized approach against one in which dimension tables are normalized. We propose a *normalized* mapping from the multidimensional to the relational model, and then compare it against the denormalized representation described in Subsection 3.1.1.

The basic idea is the following: for each functional dependency implied by the dimension hierarchy, build a table with the attributes in both sides of the dependency.

The normalized relational representation of a dimension d is a pair $(S_d, \mathcal{T}_d)$ where S_d is the schema and $\mathcal{T}_d$ the instance of the representation. S_d and $\mathcal{T}_d$ are defined below.

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

- $rname$ is $dname$.
- For each level l_i such that $l_i \preceq All$ there exists a relational schema $name(l_i, All)[l_i]$ in Sc ($name(l_i, All)$ denotes the name of the relation). Moreover, $name(l_i, All)$ is a relation of arity=1 ("All" is dropped out).
- For each pair of levels $l_i, l_j \in L$ such that $l_i \preceq l_j$, and $l_j \neq All$, there is a relational schema $name(l_i l_j)[l_i, l_j]$ in Sc (i.e., $name(l_i l_j)$ is a relation of arity=2).
- For each pair of levels $l_i, l_j \in L$ such that $l_i \preceq l_j$ there is a weak functional dependency $r : l_i \rightarrow l_j$ in $\mathcal{F}$.

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

- For each relation schema $name(l_i, All)[l_i] \in Sc$ there is a relation with the elements in $InstSet(l_i)$ as tuples.
- For each relation schema $name(l_i, l_j)[l_i, l_j] \in Sc$ we have a relation with the pairs in $\rho_{l_i}^{l_j}$ as tuples. Thus, if $\rho_{l_i}^{l_j}(a) = b$, then $< a, b > \in name(l_i, l_j)[l_i, l_j]$.

Example 19 *A normalized representation for the dimension of Figure 2.5 will have the same functional dependencies of Example 18. The relational schemas in S_c will be the following:*

$S_c = \{IB(itemId, brand)$, $IC(itemid, category)$, $BC(brand, company)$, $CO(company)$, $CAT(category)\}$.

Relations CO and CAT represent the edges $company - All$ and $category - All$, respectively.

The instance of relation $IB(itemId, brand)$ will be:

itemId	brand
$null$	b_1
i_2	b_2
i_3	b_3
i_4	b_3
i_5	b_3

3.2 Denormalized vs. Normalized Representations

In a previous work [HMV99b] we described the algorithms which compute the update operators in a ROLAP implementation, either normalized or denormalized. In that work we compared how the normalized and denormalized relational representations perform with respect to dimension updates.

3.2.1 Analytical Results

The results presented in the work mentioned above showed that, in general, *Generalize, Specialize* and *DelLevel* are more expensive in the denormalized representation. The reason for this is that in the normalized representation, a generalization or specialization reduces to the addition of a table, while in the denormalized representation a schema update of the relation is required(an attribute must be added to the table representing the dimension). The deletion of a level in the normalized representation is implemented by dropping the tables containing attributes representing such level, while in the denormalized representation table updates must be performed (implemented as a `DROP COLUMN` statement). On the other hand, relating and unrelating levels have a higher cost when the dimension is represented as a set of normalized tables, because the checking of the preconditions operates over the transitive closure of the rollup functions, which must be computed through a join of the relations representing the dimension, while in the denormalized representation, as we showed in Lemma 4, the relation itself represents the transitive closure of the set of rollup functions in the dimension. For the same reason, instance updates, either single or complex, should perform better in the denormalized representation.

3.2.2 Description of the Study

Given the theoretical results described in the previous subsection, the main objective of the experiments was studying how the complex instance update operators performed under the denormalized and normalized relational representations. The experimental results, along with the already commented theoretical ones, would define which representation adopt for our implementation. As an additional result, we wanted to confirm the intuition on the better performance of the complex instance update operators over the equivalent sequence of basic ones.

We defined six different dimension configurations, in order to study the influence of the dimension schema over the performance of the operators. These schemas ran from simple ones to complex graphs like the one depicted in Figures 3.1 and 3.2. The intention was to test the application of

the operators at different dimension levels, with different numbers of incoming and outgoing edges. The structure of the dimension influences the consistency checking the algorithms must perform. Thus, we wanted to find out how the number of conflicting levels affects the reclassify and split operators in both representations. We focused on these two operators because the merge and update operators do not require checking complex preconditions.

Normalized and denormalized relations representing the dimensions to be tested were created, and populated with synthetic data. The algorithm for generating the synthetic data works as follows: the number of elements in each dimension level (i.e. the cardinalities of the instance sets) is specified by the user. In each pass, an element i in the bottom level l_{inf} is chosen, and assigned an element j in every level l_i above l_{inf} in the hierarchy. The same step is then repeated for each l_i until the top level is reached. At each step, consistency is checked. Elements are assigned as evenly as possible, meaning that in every step, a different element in l_i is chosen if it leads to a consistent dimension. When all elements in a level were chosen, the sequence is re-started. If eventually some nodes cannot be assigned, they are discarded. The resulting dimension verifies consistency. The algorithm populates both relational representations with the same information. As we wanted to simulate real situations, the number of elements in each level decreased as we populated levels upward in the dimension hierarchy. The number of elements in the bottom level ran from five hundred to four and a half thousand.

Then, for each dimension configuration, and each dimension instance, reclassification was applied over different dimension levels(for both relational representations). For instance, considering the dimension in Figure 3.1, we reclassified an element i in level A initially assigned to element j in level B, to a different element in B. We repeated this procedure for diferent elements in A, taking the average performing time as a result.

In order to test the *split operator*, the variables we considered relevant were the cardinality of the split(the number of new elements resulting from the operator) and the number of elements currently assigned to the element being split (i.e., the elements which must be reassigned). Thus, for each dimension configuration and for each level, we chose and element and split it into different numbers of elements, and repeated the test with a different element(and a different number of assigned elements).

The chosen metric was the response time for each operator. The reason for this is that response time is a critical issue in OLAP.

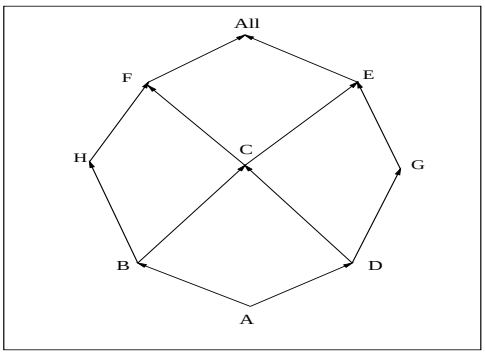

Figure 3.1: Schema of a tested Dimension

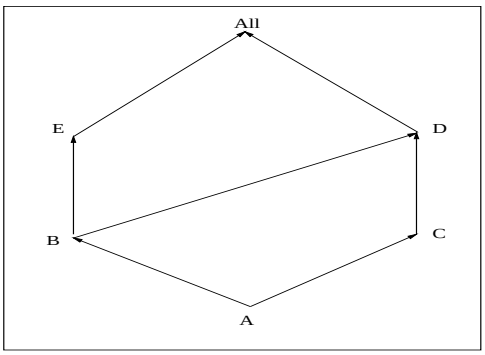

Figure 3.2: Another testing dimension schema configuration

The implementation language was Visual Basic 6.0, and the database, SQL Server 7.0 running on a Pentium III 500 Mhz desktop computer, under Windows NT 4.0 Operating System.

3.2.3 Experimental Results

We will describe the results we obtained for the *reclassify* operator applied over the dimension of Figure 3.1, and for the *split* operator applied over the dimensions of Figures 3.1 and 3.2. These results are representative of the overall experimental results. Figures 3.3 to 3.5 show the results for the *reclassify* operator. The interpretation of these charts is as follows. Let us consider for instance Figure 3.4. A point in each curve in this figure was obtained as follows: for the dimension depicted in Figure 3.1, and for an instance such that such that the number of elements in level B is indicated in the horizontal axis, several reclassifications were performed from elements in B to elements in level C, in both representations. Each point represents the average response time for these reclassifications.

We can observe that the denormalized representation performs better than the normalized one,

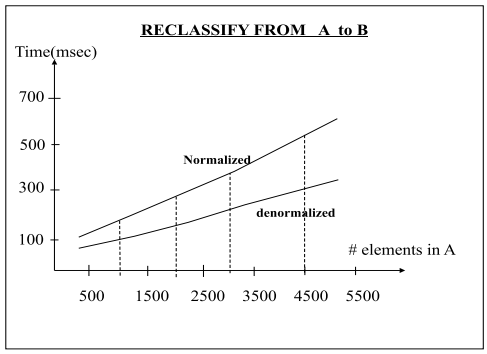

Figure 3.3: Results for Reclassify(1)

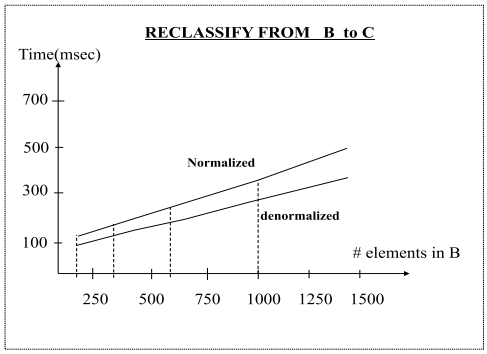

Figure 3.4: Results for Reclassify(2)

no matter the level in the graph at which the updates occur. Response times in Figure 3.5 are much better than the ones in Figures 3.3 and 3.4, and times in Figure 3.4 are slightly better than in Figure 3.3, denoting the influence of the number of elements in the levels. Also note that at higher levels in the hierarchy, the curves get closer to each other, meaning that both representations perform similarly as the number of elements in the level to be reclassified decreases.

Figure 3.6 shows results for the splitting of elements in level E of the dimension depicted in Figure 3.1. Figure 3.7 shows the results obtained splitting elements in level D in the dimension of Figure 3.2. In both cases, the results, again, favored the denormalized representation. As we explained above, the variables here are the number of elements to assign to the newly created nodes, and the number of new elements to be created. For instance, in Figure 3.6 we can see that when we split an element into 15 in level E of the dimension of Figure 3.1, and assigned 30 elements in the levels below E, the operation was completed in 1320 and 2416 milliseconds in the denormalized and normalized representations, respectively.

Finally, we compared the performance of the complex operators against equivalent sequences

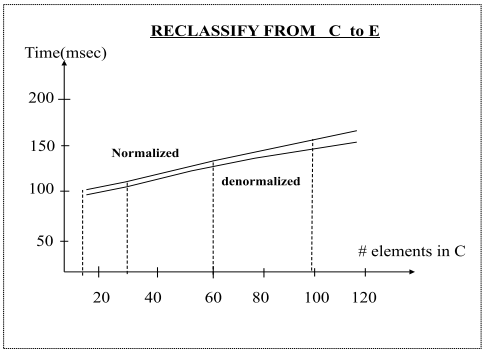

Figure 3.5: Results for Reclassify(3)

card(Split)	#elements to assign	Time(denormalized)(msec)	Time(normalized)(msec)
3	6	164	1364
5	10	272	1444
7	14	440	1600
11	22	772	1916
15	30	1320	2416

Figure 3.6: Results for Split on E - Configuration of Figure 3.1

card(Split)	#elements to assign	Time(denormalized)(msec)	Time(normalized)(msec)
3	6	220	820
5	10	552	1096
7	14	880	1436
11	22	1488	1984
15	30	2420	2860

Figure 3.7: Results for Split on D- Configuration of Figure 3.2

Type	Time(denormalized)(msec)	Time(normalized)(msec)
card(Split)= 2	108	858
Add + Del (card(Split)=2)	1524	2636
card(Split)=3	220	820
Add + Del (card(Split)=3)	1850	2926

Figure 3.8: Results for Split vs *AddInstance/DelInstance* on D

of basic ones under both representations. All the tests favored the complex operators and the denormalized representation. For the dimension of Figure 3.2 the results are shown in the table of Figure 3.8. For instance, splitting an element in level D into two new elements using the *split* operator, took 108 and 858 milliseconds in the denormalized and normalized representations, respectively. Performing the equivalent sequence of *AddInstance* and *DelInstance* operations we obtained response times of 1524 and 2636 milliseconds. Figure 3.8 shows that similar results were obtained for a split into three elements.

Based on the results presented above, in the implementation we will describe in Section 3.4 we adopted the denormalized representation.

3.3 OLEDB for OLAP and Multidimensional Expressions(MDX)

In order to allow a thorough understanding of the implementation we will present in Section 3.4, we give a very short description of OLE DB for OLAP, the multidimensional extension to Microsoft's OLE DB data access layer [Mic].

Roughly speaking, OLE DB is a set of low level interfaces allowing 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. The explosive growth of Internet led to applications accessing heterogeneous data, like multimedia data, semi-structured data and so on, requiring tools allowing to handle them efficiently. Microsoft provided OLE DB as an answer of the database industry to such requirements. In an OLE DB architecture, a *consumer* is any object consuming an OLE DB interface, while a *provider* is any software component which offers OLE DB interfaces.

OLE DB for OLAP is an OLE DB extension allowing to access and manipulate multidimensional

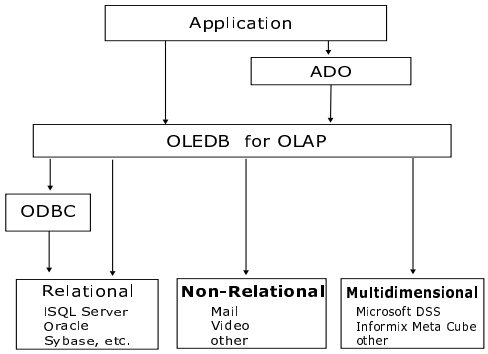

Figure 3.9: OLE DB for OLAP Architecture

data like cubes, dimensions, levels and measures, no matter how these data is physically stored. This is the reason why this standard is expected to represent for data warehousing what ODBC represents for relational databases. The OLE DB for OLAP architecture is depicted in Figure 3.9. In this figure, we can see that an application may access the data sources either via OLE DB for OLAP or via ADO, a set of high-level functions which allows users to work at a higher abstraction level.

For querying multidimensional data, OLE DB for OLAP employs *multidimensional expressions (MDX).* In this section we will just give the flavor of the language. In Section 3.4 we will show how we extended MDX in order to add statements for updating dimensions.

An MDX expression has the following basic form:

```
SELECT [<axis specification> [,<axis specification> ...]  ON COLUMNS,
       [<axis specification> [,<axis specification> ...]  ON ROWS
  FROM <cube specification>
  WHERE <slicer specification>
```

If only one dimension is required in the output, only the ``ON COLUMNS'' part is displayed in the `SELECT` clause. The cube specification in the `FROM` clause indicates the cube on which the multidimensional expression query will run. MDX supports a unique cube in each MDX query. The slicer specification on the `WHERE` clause restricts the extracting of data to a specific dimension or *member*. A *member* in MDX is a component of a dimension level. In terms of the model we introduced in Chapter 2, a *member* is an element of the *instance set* of a level. If we want to retrieve all the members in a dimension level, the `MEMBERS` keyword can be used in MDX. Also, in MDX there is a distinguished dimension called `MEASURES`, containing the measures of the cube.

For instance, let us consider the data warehouse of Example 3, and the following query over a

cube based on the fact table *DailySales* :

```
SELECT {Store.brand.MEMBERS} ON COLUMNS,
       {Time.day.MEMBERS} ON ROWS
  FROM [Sales]
  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`. Before going into the description of this statement, let us make some remarks about how the model introduced in Chapter 2 addresses dimension hierarchies.

In Example 3, the level *itemId* in dimension *Product* is related to levels *brand* and *category*, defining two ways for reaching the distinguished level *All*. We call this hierarchy a *multiple-path* hierarchy.

Definition 16 *A dimension hierarchy is called* multiple-path hierarchy *if there exist at least to different paths between l_{inf} and All. A dimension hierarchy such that there is a unique path from l_{inf} to All is called a* single-path hierarchy.

Figure 3.10 shows the MDX's `CREATE CUBE` statement for the multiple-path hierarchy of Example 3. On the other hand, Dimension *Store* in Example 3 is an example of a *single-path* hierarchy. The model we introduced in Chapter 2 supports multiple-path hierarchies in a natural way. On the contrary, MDX does only support multiple-path hierarchies in a limited way, although it is clear that these kinds of hierarchies are present in most real-life situations. Let us show this limitation of MDX by creating a cube *DailySales* based in the dimensions and the fact table of Example 3 using the `CREATE CUBE` statement. The command is depicted in Figure 3.10. Notice in this figure that, in dimension *Product*, level *itemId* belongs to two different hierarchies, in this case called *ProdCat* and *ProdBrand*. The data cube and the dimensions are populated with an `INSERT INTO` statement [Mic98]. Thus, MDX does not support the concept of rollup functions. Moreover, integrity and

```
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 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 )
```

Figure 3.10: `CREATE CUBE` statement in MDX

consistency checking is not performed. The implementation we will show in the next section, based on our multidimensional data model, gives a solution for the limitations described above.

We will not go any further into MDX description. Its complete syntax could be found in the references [Mic98, Mic].

3.4 MDDLX: an Extension to MDX

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 described in Section 3.3, 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,

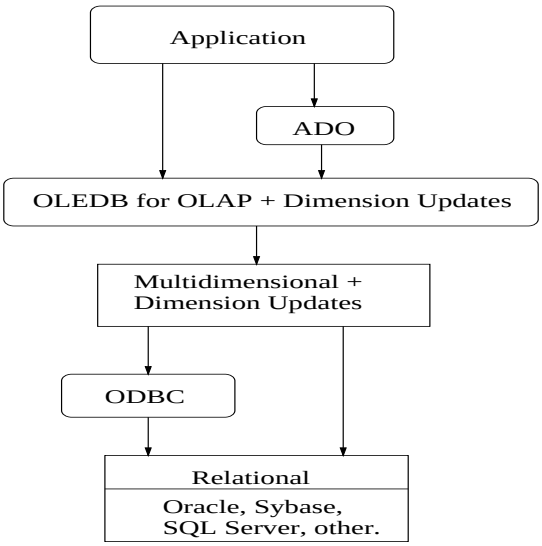

Figure 3.11: TSOLAP Architecture

without throwing away any current data.

We showed in Section 3.3 that in OLEDB for OLAP, a multiple-path hierarchy is seen as a set of single-path hierarchies, which seems to be an unnatural approach. For example, in order to build a dimension like the one depicted in Figure 2.1, we would 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.

In order to give a solution to these drawbacks, we developed an OLEDB for OLAP data provider called TSOLAP, which supports dimension updates and manages view maintenance under these updates implementing the algorithms presented in Chapter 2. In this section we discuss this implementation.

3.4.1 Architecture

The system's architecture is depicted in Figure 3.11. A multi-layered architecture was designed in which a ROLAP repository is built on top of a Relational Database, which is accessed either via ODBC or OLE DB (see Subsection 3.4.4). We introduced a layer between OLE DB for OLAP and the data source. Moreover, we extended OLE DB for OLAP with dimension update operators.

3.4.2 Data Structure

We will classify the tables in the system as *catalog* tables and *non-persistent* tables. The latter ones are tables which are created or deleted with each data cube.

- **Catalog tables.**

  Figure 3.12 depicts the data structure supporting the multidimensional repository. This catalog is stored in the same relational database as the data cube (Figure 3.11), and is accessed by the TSOLAP routines when an update or query is submitted (see Figure 3.19). We briefly describe each table in the catalog.

  - TSCUBES$. Contains a row for each one of the cubes in the system. Stores the cube name, number of dimensions and measures, and a value informing if the cube was created with the MATERIALIZE option(see below).
  - TSMeasures$. Stores a row for each measure of each cube. This row stores the data type and the kinds of aggregation for the measure (1-SUM, 2-COUNT, 3-MIN, 4-MAX, 5-AVG).
  - TSDimensions$. Contains a row for each dimension of each data cube in the system, informing name, dimension type, representation type (0-Denormalized, 1-Normalized), and the name of the dimension table, of the form *cubename_dimension$*, for example *salescube_geography$*. The dimension type allows defining different kinds of dimensions. In the present implementation, as we will explain below, the *Time* dimension has special features, so it must be distinguished from the other ones.
  - TSLevels$. Stores information about all the levels of each dimension of each cube in the system. For each dimension level l, besides the basic data, it stores the number of levels above and below l.
  - TSLevelsd$. Stores the dimension schema, i.e., every edge in the graph representing a dimension schema.
  - TSHierarchies$. Defines every hierarchy in the dimensions in the system. Recall that MDX does not support multiple hierarchies. Thus, in order to be compliant with OLE DB for OLAP, hierarchies must be exported as if they were single-path ones. For instance, attribute *hierarchy_name* is the name of each single-path hierarchy, like the ones shown in Figure 3.10.

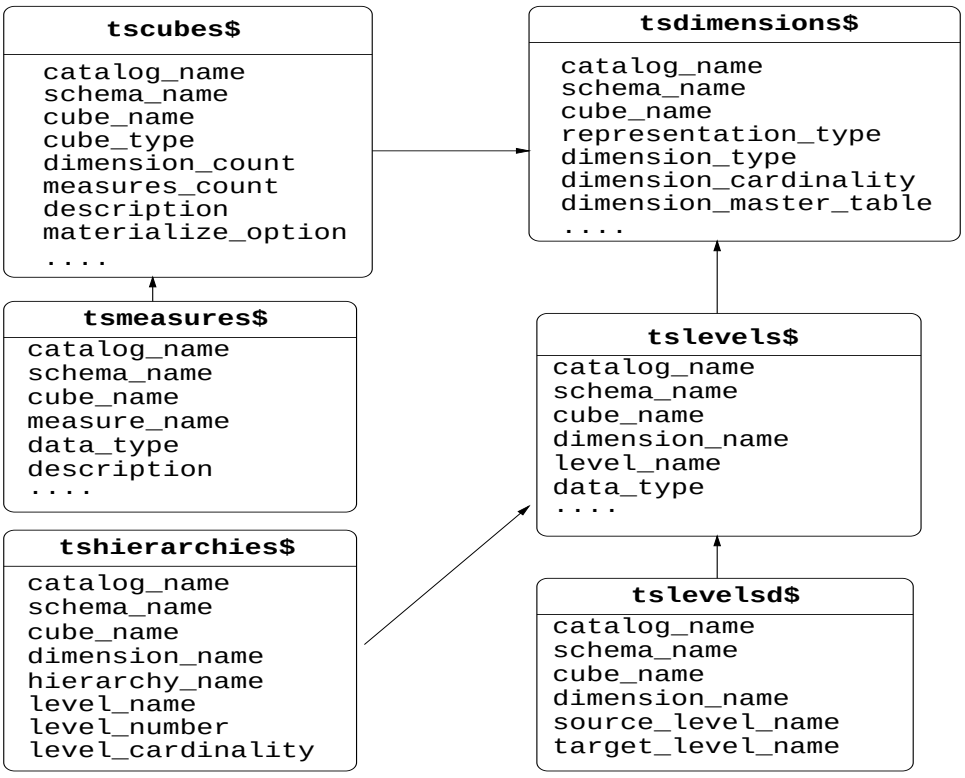

Figure 3.12: System's Catalog

- **Non-persistent tables**

  These tables are created when a data cube is created, and they are eliminated as soon as the data cube they are associated to is dropped.

  - Dimension tables. As we said above, for each denormalized dimension there is a table denoted *cubename_dimension$*. This table is created on-the-fly, and a row is inserted into the table TSDimensions$.
  - Materialized View Catalog. A catalog for materialized views is created on-the-fly for each data cube. Each catalog is denoted *vmt <cubename$>* . For instance, for a *SalesCube* data cube, the catalog for its materialized views will be denoted *vmtSalesCube$*. This table stores information about the name and the grouping level of the view. The fact table has grouping level zero.
  - Materialized View Tables. Each table name is given by the System and stored in the catalog. The table name is of the form *ts <random number$>* .

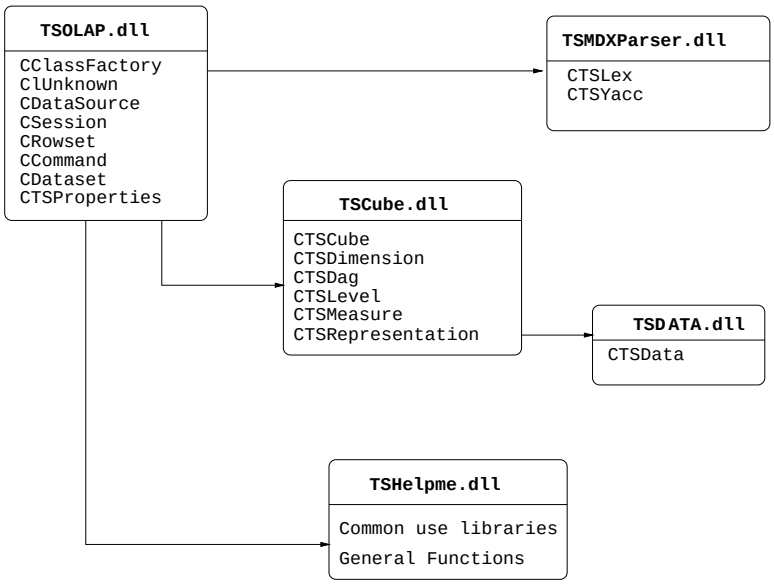

Figure 3.13: System's Libraries

3.4.3 Libraries

The software components of the system (i.e., those components which manage the data cube, the parser and the materialized views) where grouped into different libraries, depicted in Figure 3.13.

The basic functionalities needed to comply with the OLEDB for OLAP specifications were implemented in a library named TSOLAP.dll. The classes which implement the connection between a consumer and a provider are stored in this library. Thus, this class implements the methods which detect a data source, create a new session, create a new command, etc. 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, verifies if two levels are parallel, detects *conflicting levels*, etc.). CTSLex and CTSYacc implement the MDDLX parser(implemented using PCLex and PCYacc), and CTSData.dll manages the creation and maintenance of materialized views. In Figure 3.19 we show how all these libraries interact.

3.4.4 Data Access

The provider connects to a data source in order to retrieve and store all the data managed by the system. Two connection methods are supported: (a) ODBC Connection; and (b) OLE DB

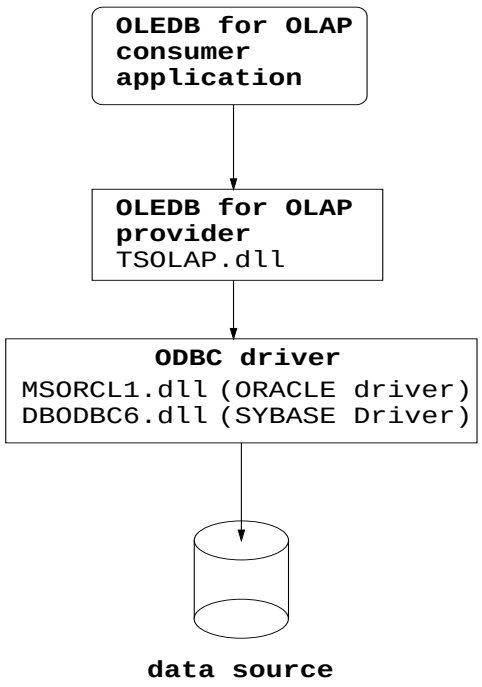

Figure 3.14: ODBC connection

Connection. Different vendors provide a wide range of ODBC drivers. Figure 3.14 shows the schema for an ODBC connection. For the OLE DB connection we used Microsoft's OLE DB providers for ODBC, MSDASQL for SQL Server and MSDAORA for ORACLE. Note in Figure 3.15 that this OLE DB driver adds a new layer, but allows to reach any DBMS via OLE DB.

3.4.5 Adding Dimension Update Support to MDX

With the support of the architecture described in the previous sections, we developed a *Multidimensional Data Definition Language* which we called *MDDLX*. It constitutes 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. We remained as faithful as possible to the formal definition of these operators. We also provide a limited SELECT statement for displaying the results, although this is not the focus of our work. The syntax is described below.

- **The SELECT statement**

```
SELECT axis_specification ON COLUMNS
FROM cube_name
WHERE slicer_specification
```

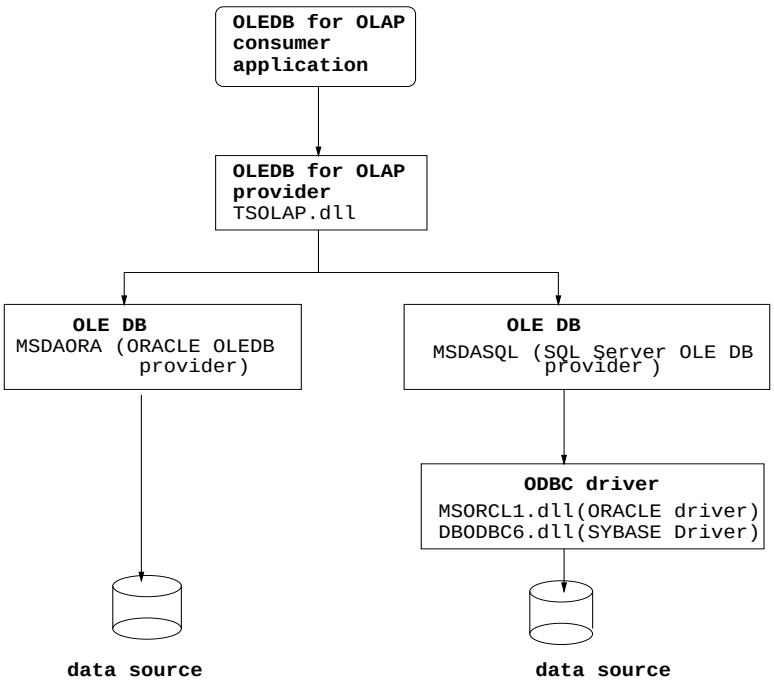

Figure 3.15: OLE DB connection

- **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]
```

In the statement above, *table_name* is a table such that each column name in it is the name of a dimension in the cube. This table is the one from which data is downloaded into the

data cube. It is assumed that this table has gone through the data extraction and cleaning processes. More, *table_name* may 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 (grouping level=0 in table *vmt* <*cubename$*> above). A new dimension is created for each `DIMENSION dimension_name` statement, such that a value in the corresponding column of *table_name* becomes a value in the bottom level of *dimension_name.* In summary, after `CREATE CUBE` is executed we will have one dimension for each `DIMENSION dimension_name` clause, each one with just one level (plus the distinguished level *All*). The other levels are created with the update statements described below.

The `TIME` dimension is generated on-the-fly, taking into account the time column in the table *table_name*, which is mandatory. Then, the base fact table will be populated with data facts in the interval defined by the clause `FROM time_value TO time_value`.

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 total number of materialized views is given by:

$$M = (\Pi_{j=1,n} K_{dj}),$$

where K_{dj} are the number of levels in each dimension d_j (including *All*).

- **Dropping data cubes**

```
DROP CUBE cube_name
```

- **Creating dimensions**

```
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.

- **Creating a $TIME$ dimension**

  `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 one 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**

  We expressed above that the `CUBE CREATION` statement creates dimensions with two levels, one of them being ALL. A dimension must be built using the dimension update statements described below, which implement the dimension update operators introduced in Chapter 2.

  - `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.

    Example 20 *The dimension* Product *in Figure 2.1 was built as follows: a dimension with levels* itemId *and* All *was initially created. Then, level* itemId *was generalized to level* category. *Finally,* brand *was 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 genitembrand
```

The table genitembrand *has the following tuples:*

genitembrand$= \{(i_1, b_1), (i_2, b_2), (i_3, b_3), (i_4, b_3)\}$.

- `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 21 *The element i_5 in Figure 2.5 is inserted with the following MDDLX command:*

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

The tuple (b_3, c_2) *indicates that* i_5 *will be assigned to brand* b_3 *and category* c_2.

- `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(see definition in Chapter 2).

Example 22 *The split operation of Example 16 is specified as follows (assume the name of the cube is* SampleCube*, and the dimension's name is* TestDim*):*

```
ALTER DIMENSION SampleCube.TestDim
    SPLIT d LEVEL D
    TO d1, d2
    FROM LEVELS (B,C)
    USING ROLLUP FUNCTIONS (splitrupBD,splitrupCD)
```

The tables splitrupBD *and* splitrupCD *have the following tuples:*

splitrupBD=$\{(b_1, d_1), (b_2, d_1), (b_3, d_2)\}$.

splitrupCD=$\{(c_1, d_1), (c_2, d_1), (c_3, d_2)\}$.

- `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

3.5 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 client tool, an OLEDB for OLAP *consumer*) called *DataSetViewer*, provided by Microsoft as part of *MDAC2.0* (Microsoft Data Access Components). In Figure 3.16 we show how a `SELECT` query is written in the *DataSetViewer* editor. The screen is split in three sections: the upper one allows 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. Figure 3.17 shows a `GENERALIZE` command. Note the log window, showing the sequence of methods of TSOLAP being invoked by the consumer (in this case, *DataSetViewer*) .

Figure 3.18 shows in detail the sequence of TSOLAP methods which the consumer must call in order to connect to the provider. These methods belong to the library TSOLAP.dll. Figure 3.19 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 the responsibilities to the objects in the way described in Section 3.4.3 and Figure 3.13.

3.5.1 Visualization: TSShow

An important issue of our work is enhancing the user's capabilities for data analysis. Thus, we developed a client tool called TSShow which allows visualization of the structure and instances of the dimensions of every cube in the System. TSShow accesses the catalog tables in order to display the system's metadata, as 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 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 3.20 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 displayed. The screen depicted in Figure 3.21 gives an idea of how TSShow shows the instances of the dimensions. In this case a *Patient* dimension is displayed (see Chapter 4).

3.6 Conclusion

In this chapter we presented TSOLAP, an implementation of the multidimensional model explained in Chapter 2, and an extension to MDX supporting dimension updates. We also described the experiments and analytical results which led to adopt a denormalized relational representation of the multidimensional model. Finally, we introduced TSShow, a visualization tool for dimensions and data cubes. In the next chapter we will apply our implementation to a case study.

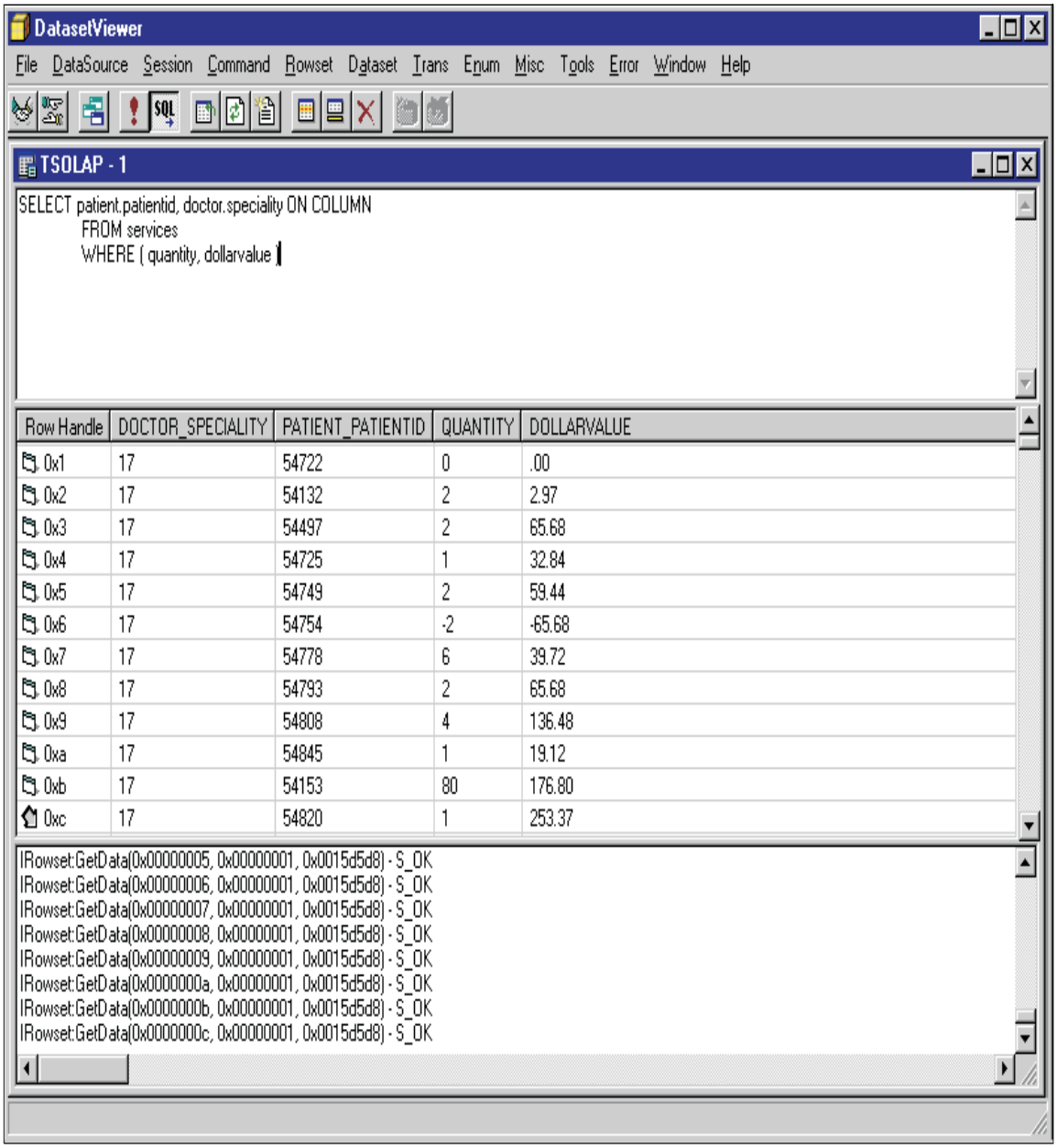

Figure 3.16: A `SELECT` query

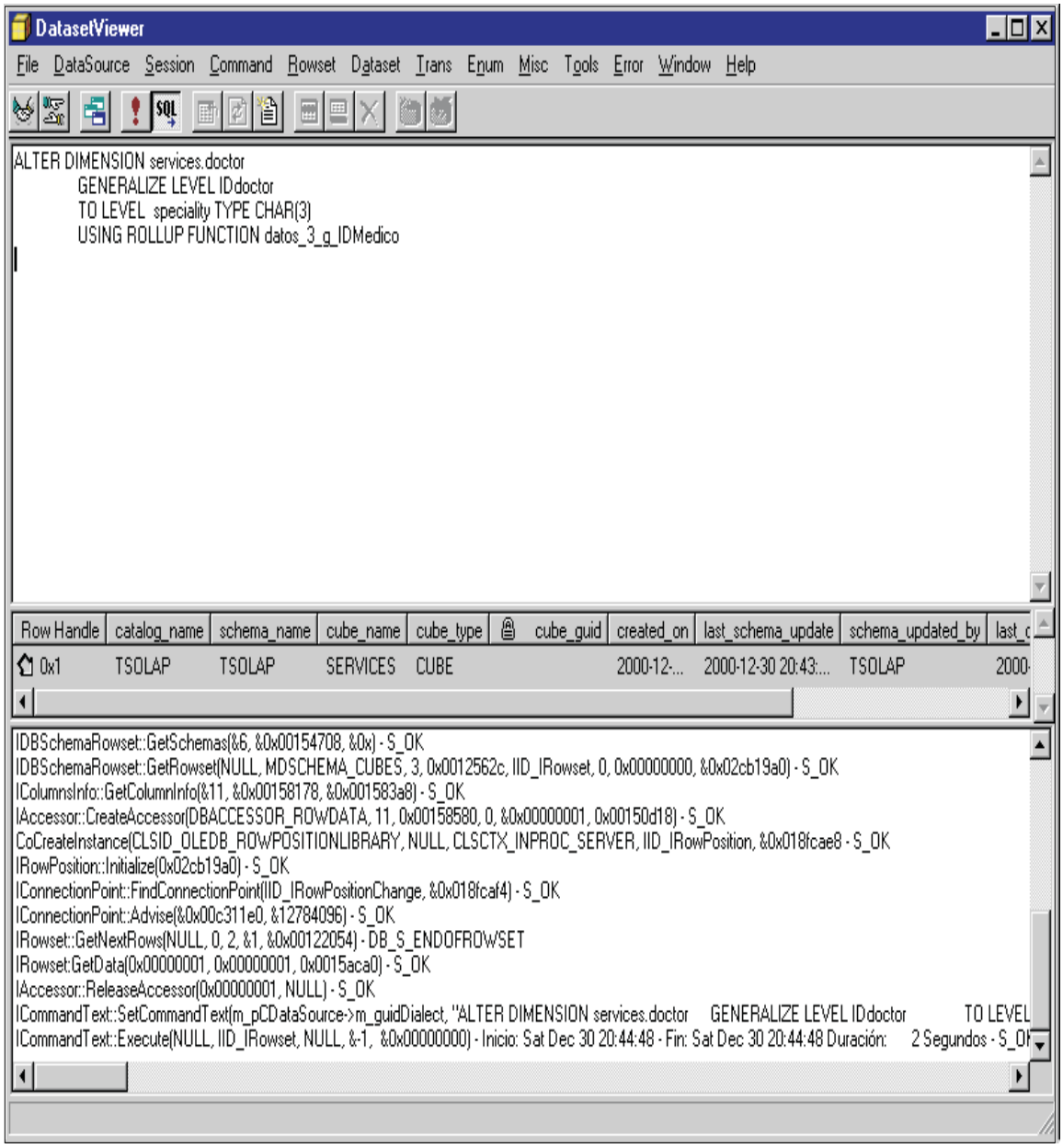

Figure 3.17: A `GENERALIZE` command

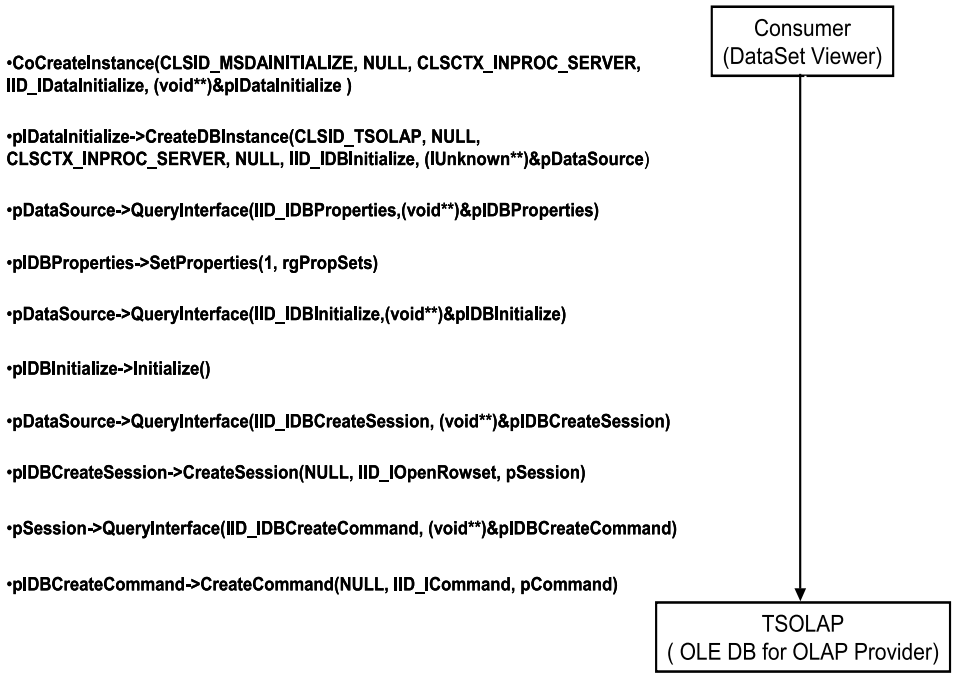

Figure 3.18: Connection Sequence

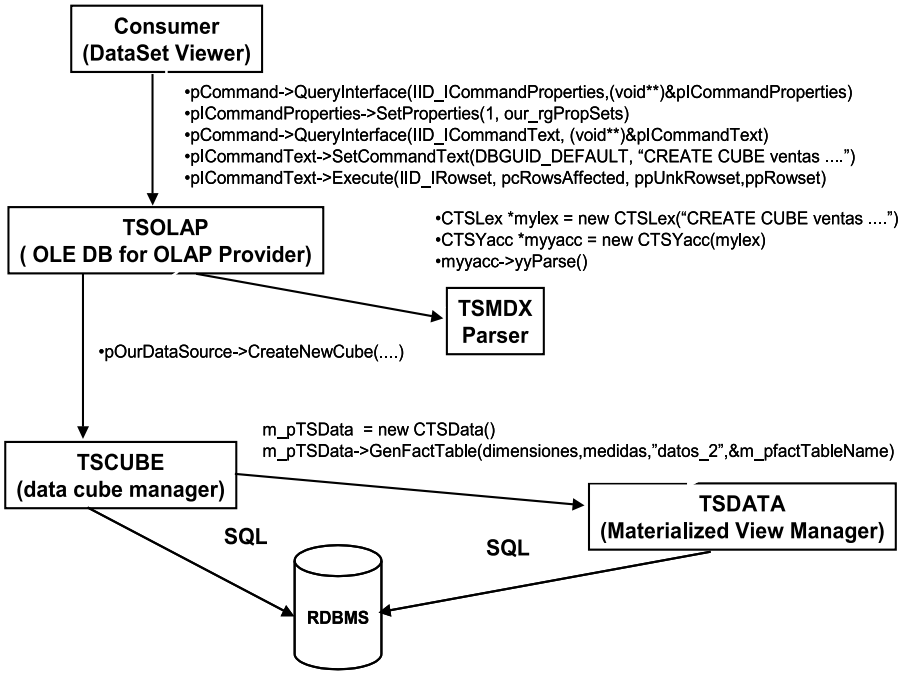

Figure 3.19: Execution steps

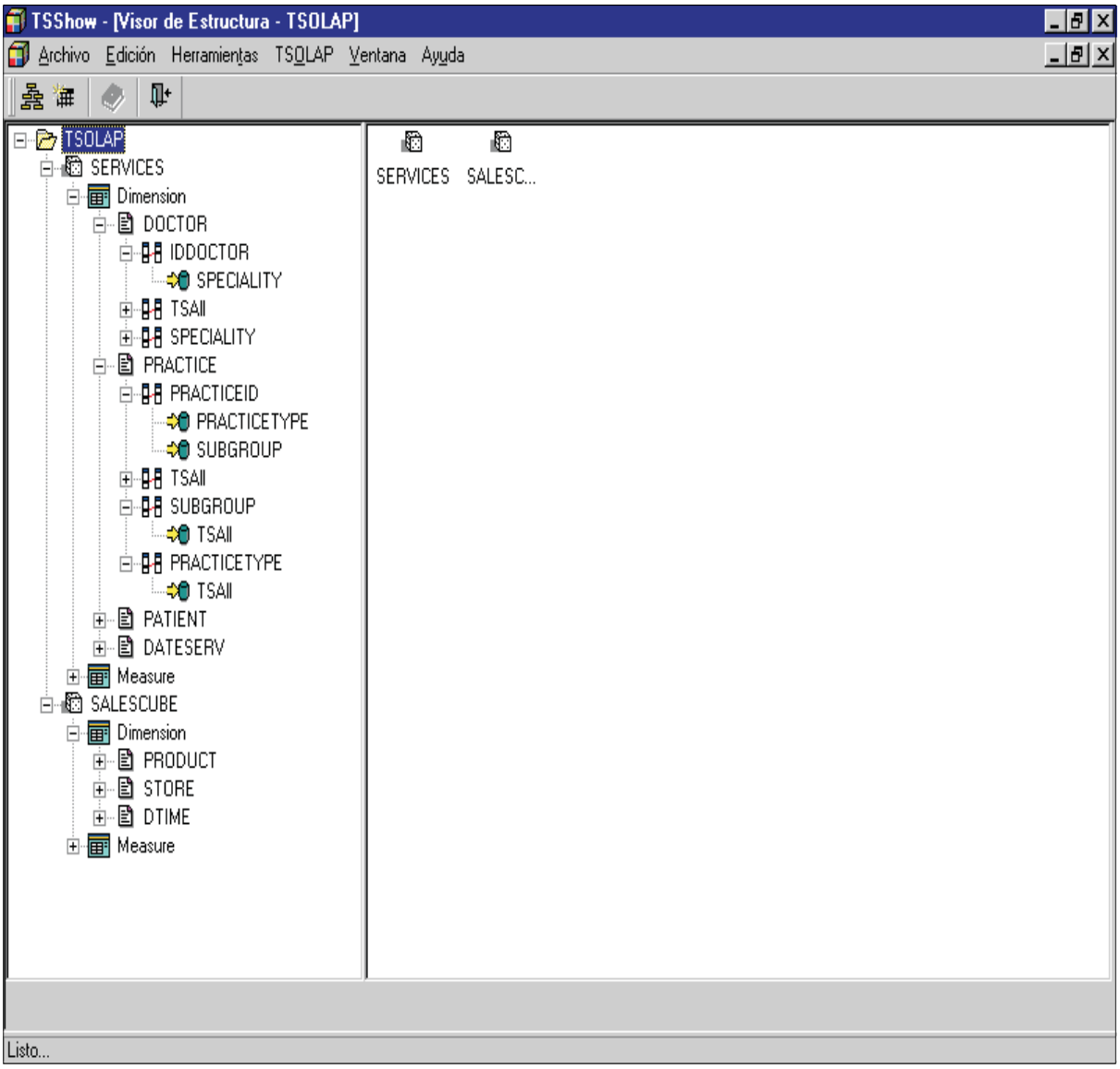

Figure 3.20: Cube and Dimension information with *TSShow*

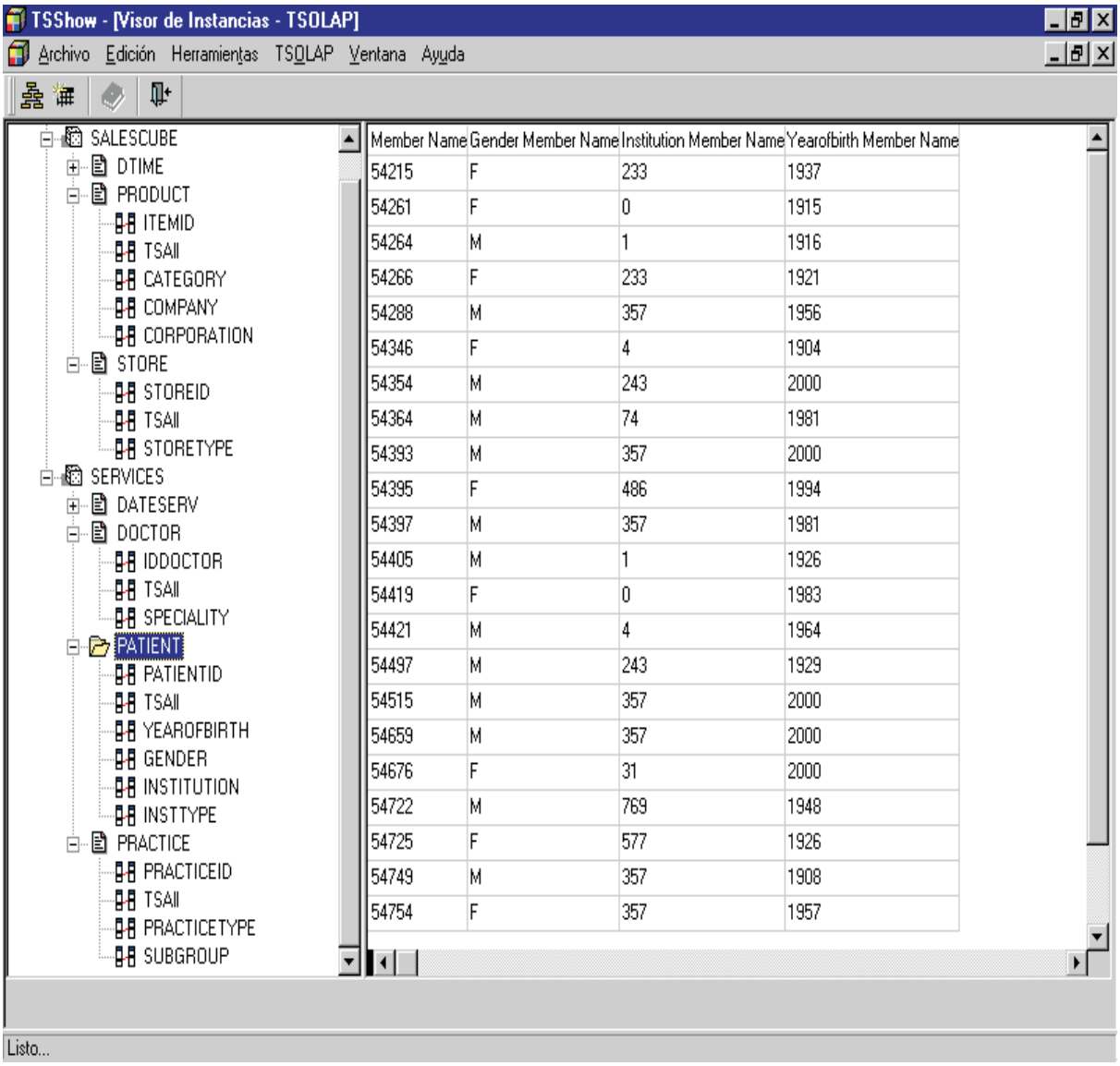

Figure 3.21: Viewing instances with *TSShow*

Chapter 4

A Case Study: A Medical Data Warehouse

In Chapter 3 we presented our implementation of the multidimensional model introduced in Chapter 2, called TSOLAP. In this chapter, we use TSOLAP in a real-life case study, a medical center in Argentina.

The chapter is organized as follows: In Section 4.1 we introduce the problem and describe how the cube and the dimensions were built. Section 4.2 discusses different ways in which TSOLAP could be applied to the case study. In Section 4.3 we establish the goals of our experiments, and the hardware we used for the tests. In Section 4.4 we present and discuss our experimental results, concluding in Section 4.5.

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 from medical treatments performed on patients being hospitalized in the clinic. Each patient gets different services, including radiographies, electrocardiograms, and so on. These services are denoted "Procedures", and are grouped into different classification levels. For instance, a procedure like "Special Radiography" is classified as follows :

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

Medicine given to a patient and disposable material are also considered "Procedures".

Data was taken from different tables in the operational database. Taking into account the available data, we designed the data warehouse as follows(see Figures 4.1 and 4.2).

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 is 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 the institution they are affiliated to. We were interested in this kind of information because it could be useful to categorize patients delivered by health insurance institutions. Moreover, these institutions are further grouped into types, for instance, private institutions, 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). This table holds data facts such that each record 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 (due to the inclusion of the `WITH MATERIALIZE` option). The created cube contains 2^n views(with n=3, the number of dimensions) with levels (procedureId, patientId,All), (All, patientId,doctorId), (procedureId,All, doctorId) and so on. Two measures are 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 4.1 and 4.2. For example, the following statement generalizes level *patientId* to level *gender*, which is of type `CHAR(1)` (an 'M' or an 'F' are stored), using the rollup function specified by the table `data3gidintengender.` This table has two columns, one named *patientId* and the other named *gender*, and each tuple stores the gender of each patient.

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

The following `RELATE` statement creates a relation between levels *practiceType* and *group.* Notice that no information is needed, except from the names of the levels 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 can be found in the Appendix.

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, we show that it is possible to add or remove elements from dimensions, or change classification levels in

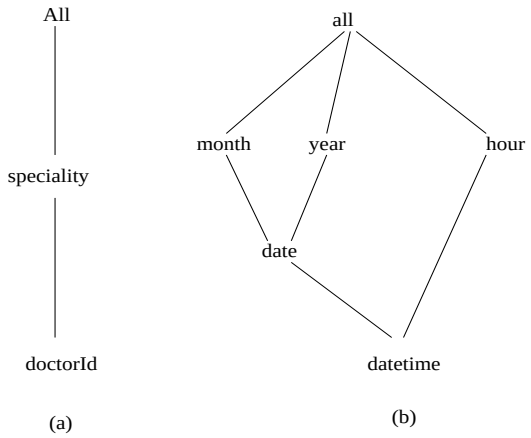

Figure 4.1: Case study: Dimensions *Doctor* and *Time*

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 performed 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 find out if there is any difference in the number of patients served, according to 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 (for instance, computer-generated patient information). Hypothetical situations could be easily addressed by replacing the actual rollup functions with the ones we wish to test (v.g., we could delete the *yearOfBirth* level and generalize again the *patientId* level, with data such that seventy percent of the patients where more than sixty years old).

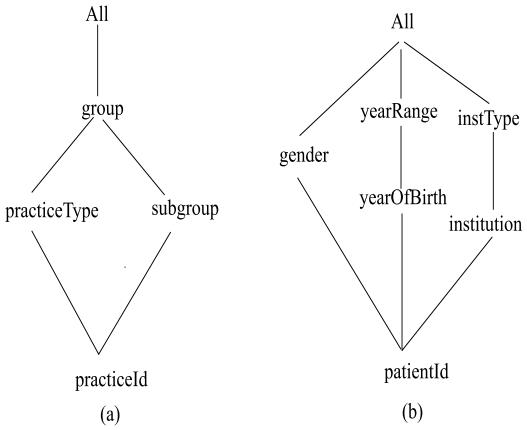

Figure 4.2: Case study: Dimensions *Procedure* and *Patient*

4.3 Objectives and Description of the Experiments

From the discussion in Section 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 Section 4.1 that we used six month of data, which involved almost 631.000 records. We partitioned this set in the six subsets shown in the table of Figure 4.3 in order to run the tests over each one of them, allowing testing the influence of the size of the data cube over the system's performance. Note that the six subsets contain the same number of patients, doctors and procedures, because we tested the operators with 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 patients, only doctors who actually delivered services before January 31st will be generalized.

In order to test the performance of the 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 following 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 for details);
- in the second sequence, we first perform all the updates over the *Time* dimension, then the

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

Figure 4.3: Data Sets

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 the number of view updates required in the second one. Both sequences can be found in the Appendix.

Our second goal was finding out the influence of view maintenance over the performance of the dimension updates. 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 to which degree an strategy with no materialization at all could affect query performance. Thus, we run 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. Further, every view is indexed on all of its columns. Actually in this study, *six hundred and thirty materialized views* are generated after all the updates are performed. We wanted to check the resulting sizes of the data cube, 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 Algorithm 3 introduced in Chapter 2 against a standard Summary-Delta-like method. We performed the tests for a `DELETE INSTANCE` update. We used three months of data(270.000 tuples in the base fact table), because we thought

that a non-optimized algorithm with the fully materialized data cube would be very inefficient. We created two data cubes: one with aggregate function `SUM`. The other with the aggregate function `MAX`. The latter allows no optimization because base data must always be accessed (recall that `MAX` is not self-maintainable with respect to deletions). 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 deleting an element in level *procedureId* over each data cube.

4.3.1 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, although we also ran our tests over an ORACLE 8.04 DBMS. For the latter we do not report results because we need further experimentation.

4.4 Experimental Results

In this section we describe the results of our experiments. We follow the order in which we stated our objectives in Section 4.3.

The tables and graphics in Figures 4.4 to 4.11 give an idea of the performance of the update operators, with respect to the number of tuples in the base fact table. Times are expressed in seconds. Figure 4.4 shows the cube creation time with total view materialization and no materialization at all. Figures 4.5, to 4.8 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 4.6 the curve corresponding to the generalization of level *patientId* is below the other two ones, while in Figure 4.7 it is above them, reflecting the influence of the number of updated views. Moreover, in Figure 4.5, both curves(corresponding to updates over *Practice*) are shifted upwards with respect to Figure 4.8, because the number of materialized

views is larger at generalization time.

Operations over 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 turns irrelevant. Figure 4.11 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 4.12.

The results presented above show that execution times are compatible with application requirements. Notice that the higher in the hierarchy the updating 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 4.13. 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 4.14 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 Algorithm 3 introduced in Chapter 2 against a non-optimized algorithm (a standard Summary-Delta-like method) for the *DelInstance* operator. Figure 4.15 shows the results, for different numbers of materialized views. We see that avoiding unnecessary joins dramatically improves performance.

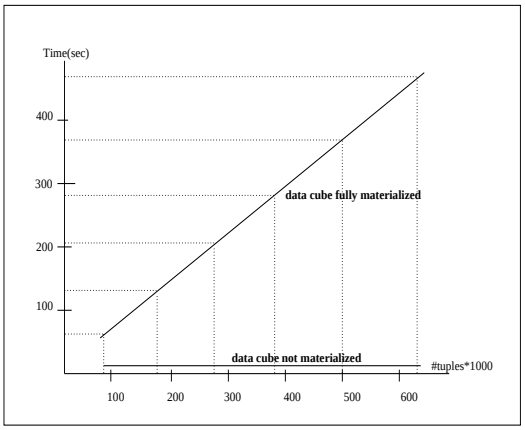

Figure 4.4: Data cube creation time

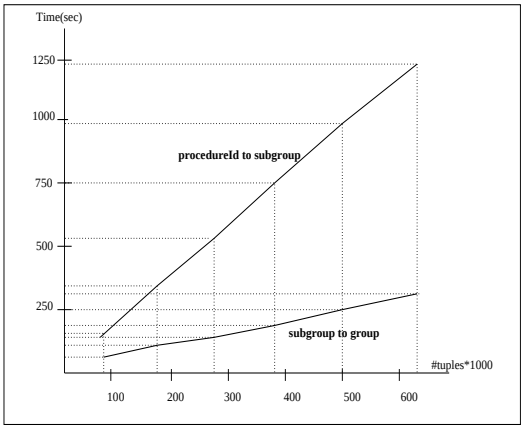

Figure 4.5: Performance results for *Generalize*(sequence 1)

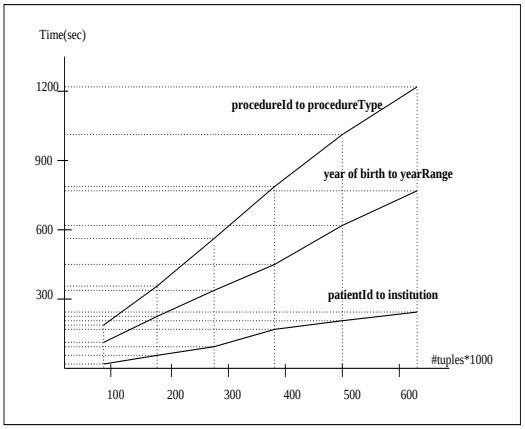

Figure 4.6: Performance results for *Generalize*(sequence 1)

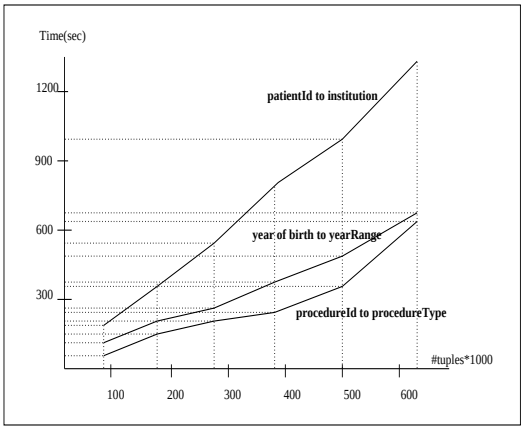

Figure 4.7: Performance results for *Generalize*(sequence 2)

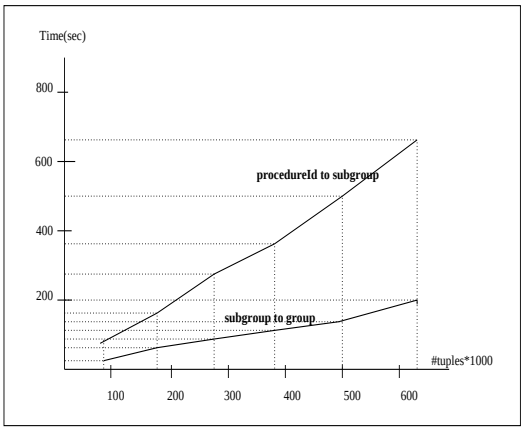

Figure 4.8: Performance results for *Generalize*(sequence 2)

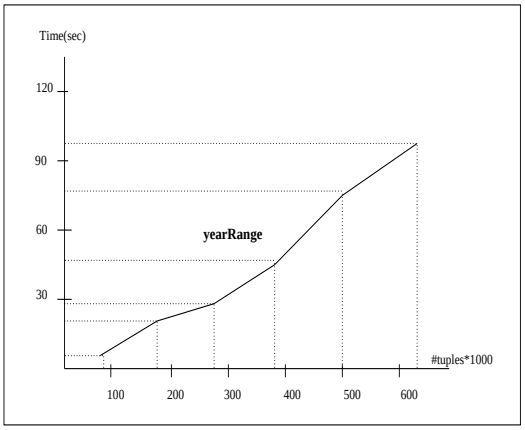

Figure 4.9: Performance results for *DelLevel*

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

Figure 4.10: Performance results for *Relate* (sequence 1)

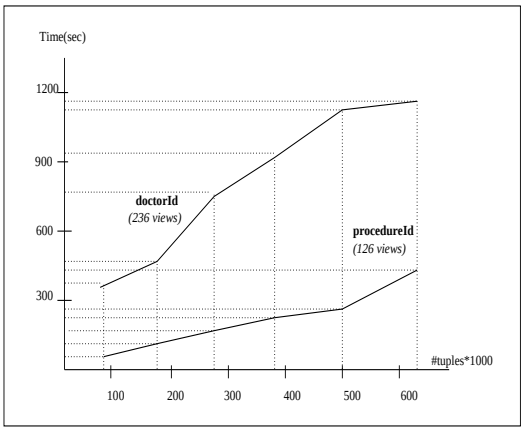

Figure 4.11: Performance results for *DelInstance*

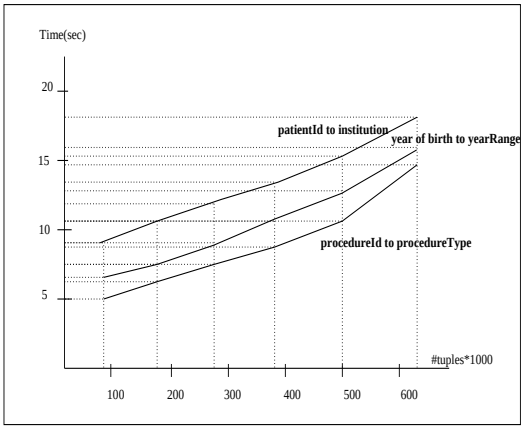

Figure 4.12: Performance results for *Generalize* with no view materialization

# of tuples in source	Time(NM)	Time	#tuples in result
90.000	≃0	16	353
160.000	≃0	26	423
230.000	≃0	40	497
370.000	≃0	68	538
600.000	≃0	120	621

Figure 4.13: 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

Figure 4.14: Data and Index disk space

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

Figure 4.15: *Del Instance* optimized vs. non-optimized

4.5 Discussion and Summary

In this chapter we applied TSOLAP to a real-life case study. The results showed that a model like the one proposed here 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.

The `DELETE INSTANCE` statement seems to be the most difficult to treat, specially when the aggregate functions of the data cube are `MAX` or `MIN` (aggregate functions which are not self-maintainable). However, we would like to remark that most of the execution time was consumed by view maintenance operations, because we were using a full materialization strategy. Moreover, computer machinery in a real working environment is much more powerful than the one used for our tests. This leads us to believe that data marts (data warehouses at department or local levels, smaller than corporate data warehouses) are the most suitable environments for our proposal. Other approaches, like partial view materialization or even no materialization at all, look promising in order to enhance the performance of dimension updates, in a way compatible with a good query performance.

In the next chapters we will embed the notion of dimension updates in a temporal database framework, leading to the concept of Temporal OLAP.

Chapter 5

Temporal OLAP

In Chapters 2 and 3 we showed that dimension data may often require to be updated on-line rather than being rebuilt from scratch every time a requirement change or an update at the data sources occur. Although the solution proposed in Chapter 2 addresses this problem, it is *memoryless*, in the sense that it loses track of the state of the dimension prior to the occurrence of the updates. In the present chapter we argue that in an evolving scenario like the above, OLAP systems need temporal features to keep track of the different states of a data warehouse throughout its lifespan and we extend the model introduced in Chapter 2.

5.1 Introduction

We will introduce the problem by means of a motivating example. For the remainder of this section we will consider dimensions as *snapshot* relations representing data as of the current time, allowing dimension updates as showed in Chapter 2. Let us suppose again, a retail data warehouse with the following dimensions: $Time, Salesperson, Customer$, $Product$. Moreover, as dimensions are organized in hierarchies, let us also assume the hierarchy $\{itemId \rightarrow itemType, itemId \rightarrow brand\}$, and the following rollup functions from $itemId$ to $itemType$: $\{(i_1, t_1), (i_2, t_1), (i_3, t_2), (i_4, t_2)\}$ (we will not be using *brand* at this time). The following table represents sales facts.

timeId	spId	customerId	itemId	salesAmount
d_1	s_1	c_1	i_1	100
d_2	s_2	c_2	i_1	100
d_3	s_1	c_3	i_3	100
d_4	s_2	c_4	i_4	100

A query asking for the *total sales per salesperson and product type*, expressed as:

```
SELECT S.spId, P.itemType, SUM(salesAmount)
FROM Sales S, Product P
WHERE S.itemId=P.itemId
GROUP BY S.spId, P.itemType
```

would return the following table:

spId	itemType	salesAmount
s_1	t_1	100
s_2	t_1	100
s_1	t_2	100
s_2	t_2	100

Suppose now that at an instant immediately after $d4$, product i_1 is assigned type t_2. A non-temporal star or snowflake schema will store $< i_1, t_2 >$, replacing the tuple $< i_1, t_1 >$, i.e., there will be no memory of the type of an item. If the user poses the same query, as all the sales occurred before the revision, she would expect to get the same result. However, she gets the following:

spId	itemType	salesAmount
s_1	t_2	200
s_2	t_2	200

What happened is that the contribution of items of type t_1 is ignored, because now all items are of type t_2.

Notice that in order to issue the query above, the user needs to know the schema of the data warehouse, that is, which are the attributes in the fact and dimension tables. However, this schema

may change over time. For instance, *itemId* may not always have been an attribute of the fact and/or dimension tables if in the early days of this data warehouse data with granularity *itemId* was not available at the sources. In this case, the query will only consider total sales made since the time at which *itemId* was added to the fact table, although information is available to obtain the total sales over the whole lifespan of the data warehouse, at least at a coarser level, like *itemType*.

As another example of inaccurate results a user could get when querying a non-temporal data warehouse, suppose that a many-to-one relationship from customers to salespersons exists, such that each salesperson is assigned a set of customers to serve. At a certain time, customer c_1, initially assigned to salesperson s_1, is reassigned to s_2. Suppose a sales manager wants to use this data warehouse to set future sales goals for each salesperson, basing the forecast on past sales data. Given the relationship between customers and salespersons, clearly this projection should be based on past volume of purchases made to customers *currently* assigned to each salesperson, no matter who was formerly assigned to whom, as a salesperson cannot expect anything from a customer no longer assigned to him/her.

We believe that new models and query languages must be developed in order to address situations like the ones commented in this section. This will avoid building ad-hoc applications, as in current commercial OLAP systems, which have no built-in temporal capabilities.

The remainder of this chapter is organized as follows: in Section 5.2 we comment on related work on temporal OLAP. In Section 5.3 we introduce the Temporal Multidimensional Data Model, and in Section 5.4 we redefine the dimension update operators of Chapter 2 in terms of the temporal data model. We conclude in Section 5.5.

5.2 Previous Work

The problem of handling "slowly changing dimensions" was mentioned by Kimball [Kim96], who suggested some partial solutions, like timestamping dimension tuples with their validity intervals. This proposal neither takes schema evolution into account, nor considers complex dimension updates.

A work carried out at the *Time Center* at the University of Arizona [BSSJ98] analyzes the performance of several SQL queries over three different approaches to the Star Schema: (a) "Time Series" fact tables; (b) "Event" fact tables; (c) Dimensions timestamped in the way proposed by Kimball. This work was, to our knowledge, the first approach to the problem of temporal OLAP.

Our work goes further, as we propose a model and a query language to address temporal issues at a higher abstraction level.

More recently, a multidimensional model for handling complex data has been introduced [PJ99], where the temporal aspect is considered as a modeling issue, and is addressed in conjunction with another data modeling problems. We are not aware of any work proposing a data warehouse evolution framework or a temporal query language for OLAP.

Recent works on maintenance of temporal views [YW98, YW00] present a view definition language operating over non-temporal data sources, along with techniques for maintaining temporal views. Although dealing with temporal databases, these works are orthogonal to our proposal, as they focus on the data sources and on how a set of temporal views are obtained and maintained, while we focus on querying a temporal multidimensional database.

5.3 The Temporal Multidimensional Model

In Chapter 2 we introduced a multidimensional model supporting dimension updates. There, dimensions were non-temporal structures. In the *Temporal Multidimensional Model* we present in this chapter, we timestamp dimension elements at the schema and instance level in order to keep track of the updates that occur during the dimension's lifespan. We also include attributes describing levels, which we intentionally omitted in Chapter 2.

In what follows we will be dealing with what in temporal databases is called *valid* time [Sno95], that is, the timestamp represents the time when the fact recorded became valid, rather than the time when it was recorded (*transaction time*). However, as a user-defined Time dimension is supported, the model can be classified as bi-temporal, like in the TSQL2 proposal(we will show this in the implementation we present in Chapter 6).

We will consider time as discrete; that is, a point in the time-line, called a *time point*, will correspond to an integer. We will also assume, except when noted, instant t_0 to be the dimension's creation instant.

5.3.1 Temporal Dimensions

The following sets are defined : a set of level names $\mathbf{L}$, where each level $l \in \mathbf{L}$ is associated with a set of values $dom(l)$; a set of attribute names $\mathbf{A}$, such that each attribute $a \in \mathbf{A}$ is associated with a set of values $dom(a)$; a set of temporal dimension names $\mathbf{TD}$; and a set of fact table names $\mathbf{F}$.

Definition 17 (Temporal Dimension Schema) *A* temporal dimension schema *is a tuple (dname, L, λ, $\preceq$, A, $\gg$, μ) where:*

- *dname* $\in$ **TD** *is the name of the temporal dimension.*
- μ *is a level in the* Time *dimension. Intuitively,* μ *defines the granularity of the dimension* dname.
- $L \subseteq \mathbf{L}$ *is a finite set of levels, which contains a distinguished level name* All, *s.t. dom*(All) = {all}. *This distinguished level is considered valid during the complete lifespan of the dimension.*
- *"*λ*" is a function with signature* $dom(\mu) \rightarrow \mathbf{L}$, *defining the instants when each level was part of the dimension.*
- *"*$\preceq$*" is a function with signature* $dom(\mu) \rightarrow 2^{\mathbf{L}\times\mathbf{L}}$, *such that for each* $t \in dom(\mu)$, $\preceq_t$ *is a relation such that* $\preceq_t^*$, *the transitive and reflexive closure of* $\preceq_t$, *is a partial order, with a unique bottom level,* $l_{inf} \in \lambda(t)$, *and a unique top level,* All, *where, for every level* $l \in \lambda(t)$, $l_{inf} \preceq_t^* l$ *and* $l \preceq_t^*$ All *hold.*
- A *is a finite set of attributes.*
- *"*$\gg$*" is a function with signature* $dom(\mu) \times \mathbf{A} \rightarrow \mathbf{L}$, *such that for every level* $l \in \lambda(t)$, a $\gg_t l$ *means that if the function is applied to an attribute* a, *it returns the level l, where attribute* a *belongs(ed) to level l at time t.*

The structure introduced by Definition 17 suggests that, given an appropriate language, it would be possible to query the schema of a dimension (using the functions $\preceq$ and $\gg$). Note that l_{inf} is unique at any time instant t, although it may not be the case at a different time. Also observe that taking a snapshot of the schema at a given time t, we are back in the non-temporal definition of a dimension schema (Definition 4). To make the former ideas more clear, let us introduce an example.

Notation In the figures of this chapter, a label t_i associated to an edge in a graph, will mean that the edge is valid for all $t \geq t_i$, and a label t_i^*, that the edge was valid for all $t < t_i$. If an edge has no label, it is valid for all the dimension's lifespan. An interval $[t_i, t_j)$ means that the edge is valid from $t \geq t_i$ to $t < t_j$.

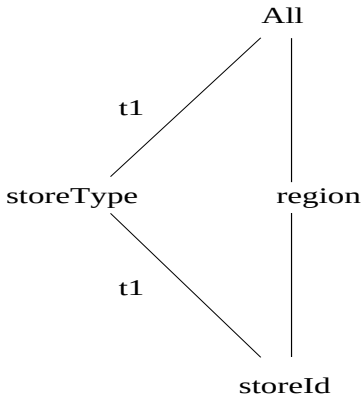

Figure 5.1: A temporal dimension schema

Example 23 *Consider a dimension* Store *where dname = Store,* $L = \{storeId, storeType, region, All\}$. *Initially(i.e. at time* t_0*) "*$\preceq$*" contains the following pairs:* $storeId \preceq_{t_0} region, region \preceq_{t_0} All$. *The addition of a new level* storeType *above level storeId at time* t_1 *will modify "*$\preceq$*" as follows:* $\preceq = \{storeId \preceq_{t_0} region, storeId \preceq_{t_1} storeType, region \preceq_{t_0} All, storeType \preceq_{t_1} All\}$. *See Figure 5.1.*

Definition 18 (Temporal Dimension Instance) *A* temporal dimension instance *is a tuple (D, TRUP, TDESC), where D is a temporal dimension schema, and:*

- *TRUP (temporal rollup) is a set of functions, satisfying the following conditions:*
  - *For every instant* $t \in dom(\mu)$, *and for each pair of levels* $l_1, l_2 \in \lambda(t)$ *such that* $l_1 \preceq_t l_2$, *there exists in TRUP a rollup function* $\rho[t]_{l_1}^{l_2} : dom(l_1) \rightarrow dom(l_2)$. *Thus, a function is defined for every snapshot taken at any instant* $t \in dom(\mu)$.
  - *For every instant t in the dimension's lifespan, and for every pair of paths* τ_1 *and* τ_2 *in the graph with nodes in* $\lambda(t)$ *and edges in* $\preceq_t$, *such that* $\tau_1 =< l_1, l_2, \ldots, l_k, l_n >$, *and* $\tau_2 =< l_1, l'_2, \ldots, l'_k, l_n >$, *we have* $\rho[t]_{l_1}^{l_2} \circ \ldots \circ \rho[t]_{l_k}^{l_n} = \rho[t]_{l_1}^{l'_2} \circ \ldots \circ \rho[t]_{l'_k}^{l_n}$.
  - *At every instant t of the dimension's lifespan, and for each triple of levels* $l_1, l_2, l_3 \in \lambda(t)$ *such that* $l_1 \preceq_t l_2$ *and* $l_2 \preceq_t l_3$,$ran(\rho[t]_{l_1}^{l_2}) \subseteq dom(\rho[t]_{l_2}^{l_3})$.
- *TDESC (temporal description) is a set of functions such that for every instant* $t \in dom(\mu)$, *and for each level* $l \in \lambda(t)$ *and for each attribute* a *such that* a $\gg_t l$, *there exists in TDESC a function with signature* $\xi[t]_l^a : dom(l) \rightarrow dom(a)$.

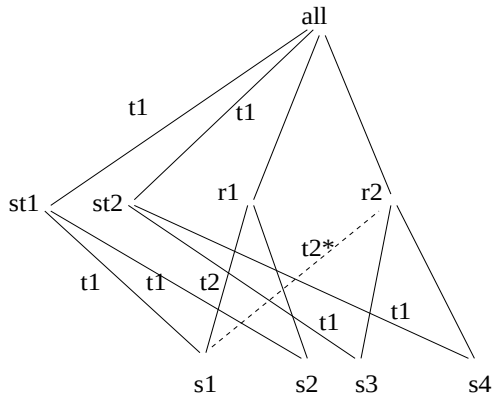

Figure 5.2: A temporal dimension instance for *Store*.

We will call the second condition in Definition 18 *snapshot consistency.*

Example 24 *Figure 5.2 shows a temporal dimension instance for dimension* Store *depicted in Figure 5.1. In this figure, we see that the rollup functions with no label are valid for the whole lifespan of* Store, *while* $\rho^{region}_{storeId}(s_1) = r_2$ *is valid until* $t_2 - 1$, *and* $\rho^{region}_{storeId}(s_1) = r_1$ *is valid from* t_2 *on.*

Definition 19 (Active Instance Set) *Given a temporal dimension* d, *a level* $l \in \mathbf{L}$, *and an instant* t, *the set of elements belonging to* $dom(l)$ *at time* t, *is called the* Active Instance Set *of* l. *We denote it* Ainstset(l,t).

Example 25 *In Figure 5.2,* Ainstset(storeId,t_2)= $\{s_1, s_2, s_3, s_4\}$. *If we delete* s_1 *at time* t_3, Ainstset(storeId,t_4)= $\{s_2, s_3, s_4\}$.

Note that in the definitions above, if the temporal functions are constant, dimensions become non-temporal, as defined in Chapter 2, allowing user-defined Time dimensions. We will come back to this issue in Chapter 6.

The previous definitions set the basis for a temporal data warehouse. Let us introduce the idea through the following example.

Example 26 *Figure 5.3 shows a sequence of updates to a temporal dimension* Product. *Initially,* Product *consisted only of level* itemId, *and the distinguished level* All. *After that, the* brand *is added to the dimension, although the initial state is not lost (Figure 5.3(b)). Later, the type of the item is inserted, with level name* itemType. *Finally, the* company *to which an item belongs is also added above level* brand *(Figure 5.3(d)).*

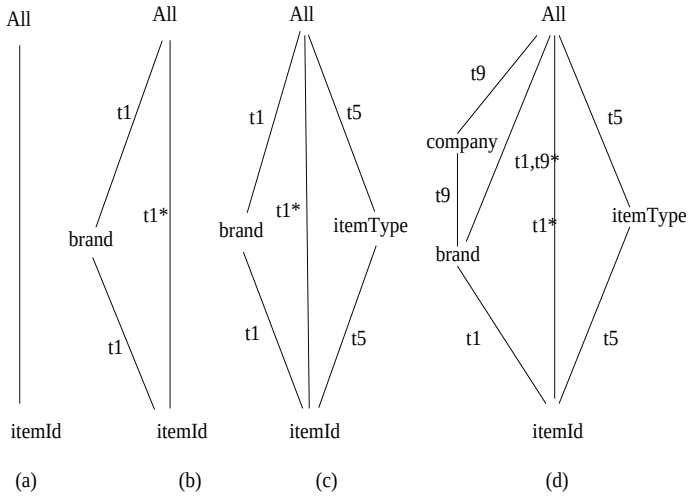

Figure 5.3: A series of updates to dimension *Product.*

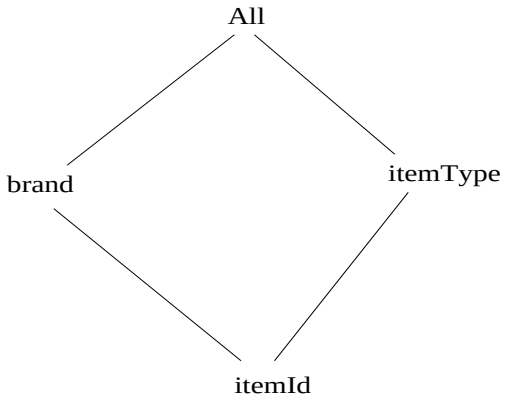

Figure 5.4: A snapshot at t_6.

A slice or snapshot(in temporal database terminology) of a dimension, taken at a given instant t, defines the state of the dimension at that time. For instance, Figure 5.4 shows a snapshot of dimension *Product* at t_6.

Definition 20 (Temporal Fact Table) *A* temporal fact table schema *is a tuple* $s = (fname,\ f, m, \mu)$*, where* m *is a level name, called the* measure *of the fact table,* μ *is a level in the* Time *dimension, and* f *is a function with signature* $dom(\mu) \rightarrow 2^{\mathbf{L}}$.

Given a temporal fact table schema $(fname, f, m, \mu)$*, a set of levels* L *in the range of* f*, and a level* μ *in the* Time *dimension, a mapping from each level* $l_i \in (L \cup \mu)$ *to* $dom(l_i)$ *is called a* point.

Given a temporal fact table schema $s = (fname, f, m, \mu)$*, a* temporal fact table instance *over it, is a partial function named* $fname$ *which maps points of* s *to elements in* $dom(m)$.

Definition 21 (Temporal Base Fact Table) *Given a set* $\mathbf{D}$ *of temporal dimensions, a* temporal base fact table *is a temporal fact table with schema* $(fname, f_D, m, \mu)$*, where* f_D *is a function with signature* $dom(\mu) \rightarrow 2^{\mathbf{L}}$*, such that for each* $t \in dom(\mu)$*, every level in* $f_D(t)$ *is a bottom level of the dimension it belongs to. Thus, a temporal base fact table is a temporal fact table such that its attributes are the bottom levels of each one of the dimensions in* $\mathbf{D}$*.*

Example 27 *Given* $D = \{Store, Product\}$ *where dimensions* Store *and* Product *are the ones of Figures 5.1 and 5.3 respectively, the Temporal Base Fact Table associated to* D *would have* $f_D(t) = \{storeId, itemId\}$*. If updates occur such that at time* t_{12}, brand *becomes the bottom level of dimension* Product, $f_D(t_{12}) = \{storeId, brand\}$.

Definition 22 (Temporal Multidimensional Database)

- *A temporal multidimensional database schema, denoted* $\mathbf{B_s}$*, is a pair* $(\mathbf{D_s}, \mathbf{F_s})$*, where* $\mathbf{D_s}$ *is a set of temporal dimension schemas, and* $\mathbf{F_s}$ *is a set of temporal fact table schemas.*
- *A* multidimensional database instance $I(\mathbf{B})$*, is a tuple* $(\mathbf{F_I}, \mathbf{D_I})$*, where* $\mathbf{D_I}$ *and* $\mathbf{F_I}$ *are dimension and fact table instances defined as above.*

In the next section we will show how the non-temporal update operators defined in Chapter 2 for non-temporal dimensions, can be extended to support temporal dimensions, preserving the dimension's history.

5.4 Temporal Dimension Updates

We will now define a set of temporal update operators allowing to perform updates over the schema or an instance of a given dimension without losing historical information. These operators are the temporal extension of the ones introduced in Chapter 2. We will not address operators over level attributes. It is trivial to define operators allowing updating an attribute belonging to a dimension level.

Notation We will consider the set L of level names in a dimension schema at any time instant t, as the union of two disjoint subsets, $L_A(t) = \{l | l \in \lambda(t)\}$, and $L_I(t) = \{l | l \in L \wedge l \notin \lambda(t)\}$.

For notation conciseness, a time point immediately before a given instant t is denoted t^- (i.e. $t^- = t - 1$).

We will give the formal definitions for the temporal version of each operator introduced in Section 2.3. In the following definitions we assume that the values for the functions $\preceq_t$ and ρ remain constant until an operator updates them.

5.4.1 Basic Temporal Structural Updates

The *Temporal Generalize* operator creates a new level, l_n, to which a pre-existent one, l, rolls up at time t. A function f must be defined from the active instance set of l, to the domain of l_n. The main difference with the non-temporal *Generalize*, is that no physical deletion occurs as a consequence of the update.

Operator 12 (Temporal Generalize) *Given a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP)$*, an instant* t*, two levels* $l \in L_A(t^-)$ *and* $l_n \notin L_A(t^-)$*, and a function* $f_l^{l_n} : Ainstset(l, t^-) \rightarrow dom(l_n)$*,* $TGeneralize(d, l, l_n, f_l^{l_n}, t)$ *updates the elements in* d *in the following way:*

- $L = L \cup \{l_n\}$;
- $\lambda(t) = \lambda(t^-) \cup \{l_n\}$;
- $\preceq_t = \preceq_{t^-} \cup \{(l, l_n), (l_n, All)\} \setminus \{(l, All)\}$;
- *The following functions are added to TRUP:*
  - $\rho[t]_l^{l_n} = f_l^{l_n}$;
  - $\rho[t]_{l_n}^{All}(e) = all \quad \forall e \in ran(f_l^{l_n})$;
  - $\rho[t]_l^{All} = NIL$;
  - $\rho[t]_{l_i}^{l_j} = \rho[t^-]_{l_i}^{l_j}$*, for all other levels* $l_i, l_j \in L_A(t)$.

Example 28 *Figure 5.3 in Section 5.3 shows a series of* Generalize *operations. For instance, Figure 5.3(b) depicts the operation* $Generalize(Product, itemId, brand, f_{itemId}^{brand}, t_1)$.

The *Temporal Relate* operator defines a rollup function between two independent levels belonging to a dimension, at time t. Conditions for the temporal relate are analogous to the ones defined for the non-temporal relate (see Section 2.3.1).

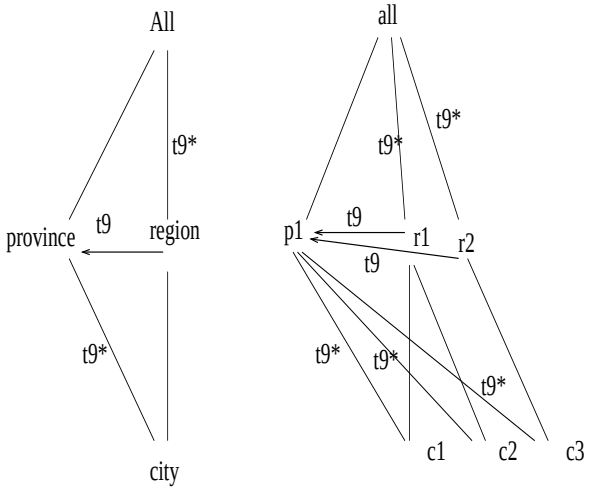

Figure 5.5: Relating regions and provinces.

Operator 13 (Temporal Relate Levels) *Given a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP)$, *a time instant* t, *a pair of levels* $l_a, l_b \in L_A(t^-)$, *such that there exists a consistency function* $f_{l_a}^{l_b}$ *between* $Ainstset(l_a, t^-)$ *and* $Ainstset(l_b, t^-)$, $TRelate(d, l_a, l_b, t)$ *updates the elements in* d *in the following way:*

- $\lambda(t) = \lambda(t^-)$;
- $\preceq_t = \preceq_{t^-} \cup \{(l_a, l_b)\} \setminus \{(l_i, l_b) | l_i \preceq^*_{t^-} l_a \wedge l_i \preceq_{t^-} l_b\} \setminus \{(l_a, l_k) | l_a \preceq_{t^-} l_k \wedge l_b \preceq^*_{t^-} l_k\}$;
- *The following functions are added to* $TRUP$:
  - $\rho[t]_{l_a}^{l_b} = f_{l_a}^{l_b}$;
  - $\rho[t]_{l_i}^{l_j} = NIL, \quad if \quad (l_j = l_b \wedge l_i \preceq^*_{t^-} l_a) \vee (l_i = l_a \wedge l_b \preceq^*_{t^-} l_j)$;
  - $\rho[t]_{l_i}^{l_j} = \rho[t^-]_{l_i}^{l_j}$, *for all other levels* $l_i, l_j \in L_A(t)$.

Example 29 *Figure 5.5 shows a typical* Geography *dimension in which at time* t_9, *it is decided that every region must belong to a unique province. As the preconditions are met,* $TRelate(Geography, region, province, t_9)$ *can be performed.*

Operator 14 (Temporal Unrelate Levels) *Given a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP)$, *a time instant* t, *a pair of levels* $l_a, l_b \in \{L_A(t^-) \setminus All \setminus l_{inf}^{t^-}\}$, *s.t.* $l_a \preceq_t l_b$, $TUnrelate(d, l_a, l_b, t)$ *updates the elements in* d *in the following way:*

- $\lambda(t) = \lambda(t^-)$;
- $\preceq_t = \preceq_{t^-} \setminus \{(l_a, l_b)\} \cup \{(l_i, l_b) | l_i \preceq_{t^-} l_a \wedge l_i \not\preceq^*_{t^-_{ab}} l_b\} \cup \{(l_a, l_k) | l_b \preceq_{t^-} l_k \wedge l_a \not\preceq^*_{t^-_{ab}} l_k\}$;

  $l_i \not\preceq^*_{t^-_{ab}} l_j$ *means that no path exists at time* t^- *between* l_i *and* l_j *which includes* l_a, l_b. [1]
- *The following functions are added to TRUP :*
  - $\rho[t]^{l_b}_{l_a} = NIL$;
  - $\rho[t]^{l_b}_{l_i}(x) = y \quad if \quad \rho[t^-]^{l_a}_{l_i}(x) = z \wedge \rho[t^-]^{l_b}_{l_a}(z) = y$, *and* $l_i \preceq^*_{t^-_{ab}} l_b$;
  - $\rho[t]^{l_j}_{l_a}(x) = y \quad if \quad \rho[t^-]^{l_b}_{l_a}(x) = z \wedge \rho[t^-]^{l_j}_{l_b}(z) = y$ *and* $l_a \preceq^*_{t^-_{ab}} l_j$;
  - $\rho[t]^{l_j}_{l_i} = \rho[t^-]^{l_j}_{l_i}$, *for all other levels* $l_i, l_j \in L_A(t^-)$.

If $l_a = l^{t^-}_{inf}$ *(in what follows* l_{inf}*) or if* $l_b = All$, *the following holds:*

- $\text{TUnrelate}(d, l_{inf}, l_b, t) = \text{TRelate}(d, l^{t^-}_c, l_b, t)$, *where* $l^{t^-}_c$ *is the level closest to* l_a *such that* $l_b \parallel l^{t^-}_c$ *and* $\text{TRelate}(d, l^{t^-}_c, l_b, t)$, *is possible.*

  By closest level we mean that there is no path $< l_{inf}, \ldots l_j \ldots l^{t^-}_c >$ *such that* $\text{TRelate}(d, l_j, l_b, t)$, *is possible. If there exist two such levels within the same distance from* l_{inf}, *anyone can be chosen.*
- $\text{TUnrelate}(d, l_a, All, t) = \text{Relate}(d, l_a, l^{t^-}_d, t)$, *where* $l^{t^-}_d$ *is the level closest to All such that* $l^{t^-}_d \parallel l_a$ *and* $\text{TRelate}(d, l_a, l^{t^-}_d, t)$, *is possible.*

  By closest level we mean that there is no path $< l^{t^-}_d, \ldots l_j \ldots All >$ *such that* $\text{TRelate}(d, l_a, l_j, t)$, *is possible. If there exist two such levels within the same distance from* l_{inf}, *anyone can be chosen.*

The *Temporal Delete Level* operator deletes a level and its rollup functions (in the temporal database sense). As in the non-temporal case, the level to be deleted cannot be the lowest one in the dimension at the time of the deletion (l_{inf}), unless it rolls up to only one higher level. In this case, the fact tables associated with the dimension must be updated(see Example 30).

Operator 15 (Temporal Delete Level) *Given a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP)$, *a time instant* t, *a level* $l \in L_A(t^-), l \neq All$ *such that if* $l = l_{inf}$, *then there is only one level* l_j *such that* $l \preceq_{t^-} l_j$, $TDelLevel(d, l, t)$ *updates the elements in* d *in the following way:*

[1] The last conditions prevent the addition of the arcs in case alternative paths (not including l_a, l_b,) between (l_i, l_b) or (l_a, l_j) existed in $\preceq$ at time t^-.

- $\lambda(t) = \lambda(t^-) \setminus \{l_n\}$;
- $\preceq_t = \preceq_{t^-} \setminus \{(l_1, l_2) | (l_1 = l) \vee (l_2 = l)\} \cup \{(l_1, l_2) | (l_1 \preceq_{t^-} l) \wedge (l \preceq_{t^-} l_2) \wedge (l_1 \not\preceq^*_{t^-} l_2)\}$[2];
- *The following functions are added to TRUP* :
  - $\rho[t]^{l_j}_{l_i} = NIL \quad if \quad l_j = l \vee l_i = l$;
  - $\rho[t]^{l_j}_{l_i}(x) = y \quad if \quad \rho[t^-]^{l}_{l_i}(x) = z \wedge \rho[t^-]^{l_j}_{l}(z) = y$;
  - $\rho[t]^{l_j}_{l_i} = \rho[t^-]^{l_j}_{l_i}$, *for all other levels* $l_i, l_j \in L_A$.

5.4.2 Complex Temporal Structural Updates: TSpecialize

The *Temporal Specialize* operator updates the bottom level of a dimension d at an instant t. A new level l_n is added below the lowest level of d, l_{inf}, s.t. $l_{inf} \in L_A(t^-)$, becoming the new current lowest level of the dimension. Again, a function must be defined for this rollup. This is a key operation, because the base fact tables in a multidimensional database **B** involving d, must be updated in consequence.

Operator 16 (Temporal Specialize) *Given a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP)$, *a time instant* t, *a level* $l_n \notin L_A(t^-)$, $l_{inf} \in L_A(t^-)$ *(the bottom level of d) and a function* $f^{l_{inf}}_{l_n} : dom(l_n) \rightarrow Ainstset(l_{inf}, t^-)$, $TSpecialize(d, l_n, f^{l^t_{inf}}_{l_n}, t)$ *updates the elements in d in the following way:*

- $L = L \cup \{l_n\}$
- $\lambda(t) = \lambda(t^-) \cup \{l_n\}$;
- $\preceq_t = \preceq_{t^-} \cup \{(l_n, l^t_{inf})\}$;
- *The following functions are added to TRUP:*
  - $\rho[t]^{l_{inf}}_{l_n} = f^{l_{inf}}_{l_n}$;
  - $\rho[t]^{l_j}_{l_i} = \rho[t^-]^{l_j}_{l_i}$, *for all other levels* $l_i, l_j \in L_A(t^-)$.

Example 30 *Suppose the following situation, depicted in figure 5.6. Before* t_1, *each salesperson was assigned customers in a unique city. At time* t_1, *level* idSalesperson *was dropped from the*

[2] this last condition implies that no alternative paths between (l_1, l_2) exist at the time of the deletion.

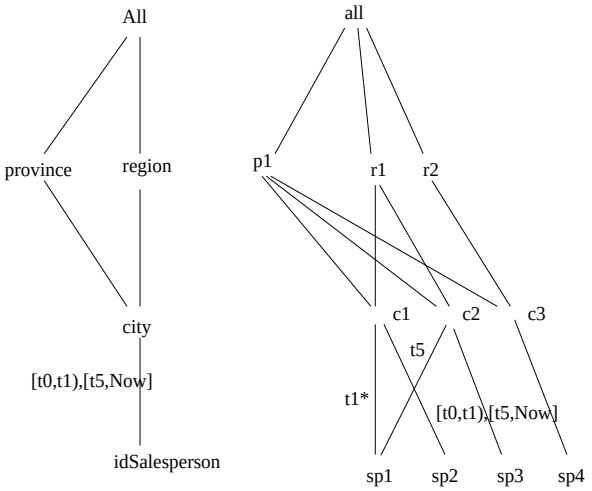

Figure 5.6: A Temporal dimension before and after Specialization.

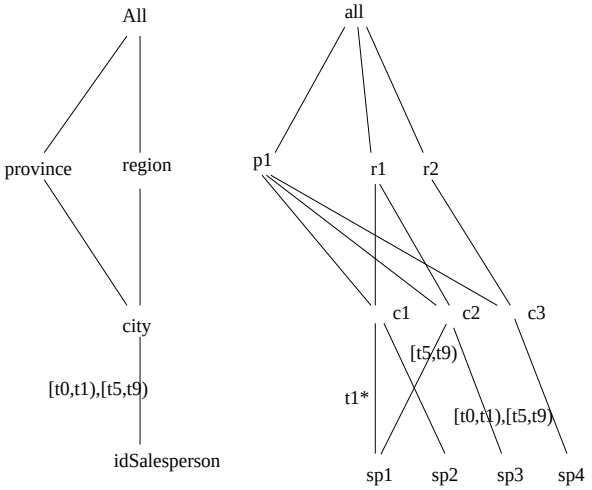

Figure 5.7: Deleting level IdSalespersons at time t_9.

dimension, but at time t_5, this decision is reversed. Thus, at t_5, a specialization occurred. However, note that the distribution of cities changed in a way such that salesperson sp_1 was moved to city c_2. Salespersons sp_1, sp_2 and sp_3, remain assigned to the same cities as before t_1, denoted in the figure as $[t_0, t_1), [t_5, Now]$.

As an example of the TDelLevel *operator, suppose at time t_9, salespersons are not needed any more in the dimension of Figure 5.6. Thus, level* idSalesperson *is deleted. Obviously, no new edge is created, because* idSalesperson *was the bottom level at time t_8. The dimension of Figure 5.6 after the temporal deletion of level* idSalesperson *is depicted in Figure 5.7.*

5.4.3 Basic Temporal Instance Updates

We will define now temporal operators allowing modifications to the instance of a dimension. Again, no element will be deleted, but timestamped, indicating that the element is no longer active.

The *Temporal Add Instance* operator inserts a new element, say x, at an instant t into a level $l_a \in L_A(t^-)$. The element must not exist in the Active Instance Set of l_a at time t^-. As in the non-temporal model, the operator must be provided with the pairs (l_i, y), such that every $l_i \in L_A(t^-)$ is a level to which l_a directly rolls up($l_a \preceq_{t^-} l_i$), and y belongs to the Active Instance Set of l_i at time t^-. Thus, y is the element in level l_i to which x will roll up.

Operator 17 (Temporal Add Instance) *Assume a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP)$, *a time instant* t, *a level* $l_a \in L_A(t^-)$, *an element* $x_a \in dom(l_a)$, $x_a \notin Ainstset(l_a, t^-)$, *a set of pairs* $P = \{(l_1, x_1), \ldots (l_n, x_n)\}$ *such that:*$l_i \in L_A(t^-)$, $x_i \in Ainstset(l_i, t^-)$, $dom(P) = \{l_i \mid l_a \preceq_{t^-} l_i\}$, *and for each pair* $(l_k, x_k), (l_s, x_s) \in P$ *such that exists a level* $l \in L_A(t^-), l_k \preceq^*_{t^-} l$ *and* $l_s \preceq^*_{t^-} l$, *the following holds:* $\rho[t^-]^l_{l_k}(x_k) = \rho[t^-]^l_{l_s}(x_s)$. $TAddInstance(d, l_a, x_a, P, t)$ *updates the rollup functions in* d, *in the following way :*

- $\rho[t]^{l_j}_{l_a}(x_a) = x, \quad if \quad (l_j, x) \in P$;
- $\rho[t]^{l_j}_{l_i} = \rho[t^-]^{l_j}_{l_i}$, *for all other levels* $l_i, l_j \in L_A(t)$.

The *Temporal Delete Instance* operator deletes, at time t an element belonging to the active instance set of a level l_a. It is only defined when no element of any level l_i such that $l_i \preceq_{t^-} l_a$, rolls up to the element being deleted.

Operator 18 (Temporal Delete Instance) *Given a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP)$, *a time instant* t,, *a level* $l_a \in L_A(t^-)$, *an element* $x_a \in Ainstset(l_a, t^-), x_a \notin \bigcup_{l \in L_A(t^-)} ran(\rho[t^-]^{l_a}_l)$, $TDelInstance(d, l_a, x_a, t)$ *updates the rollup functions in* d *in the following way :*

- $\rho[t]^{l_j}_{l_a}(x_a) = NIL, \forall l_j \quad s.t. \quad l_a \preceq_{t^-} l_j$;
- $\rho[t]^{l_j}_{l_i} = \rho[t^-]^{l_j}_{l_i}$, *for all other levels* $l_i, l_j \in L_A$.

Example 31 *Suppose a new item* i_5 *is added to the instance of* Product *represented in Figure 5.8, at time* t_1, *by means of* $TAddInstance(Product, itemId, i_5, \{(brand, b_3), (itemType, c_2)\}, t_1)$. *Figure 5.8(a) shows the result. After that, at time* t_2, *we delete item* i_1 *from level* itemId *(the bottom*

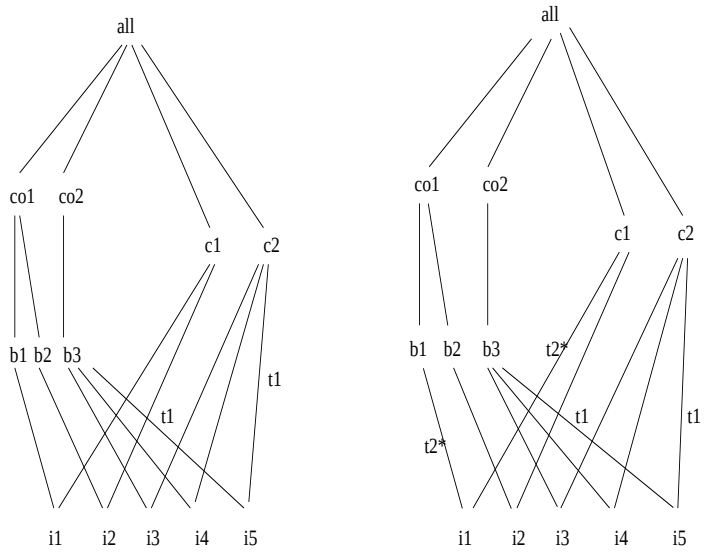

Figure 5.8: (a)Temporal Add instance (b)Temporal Delete instance.

level of Product *at that time). The operation is invoked as* $TDelInstance(Product, itemId, i_1, t_2)$. *See figure 5.8(b). The reader is suggested to compare Figures 2.5 and 5.8 which present the same situation in the non-temporal and temporal approaches, respectively.*

5.4.4 Complex Temporal Instance Updates

Where possible, we will avoid repeating concepts already present in Chapter 2.

Operator 19 (Temporal Reclassify) *Given a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP)$, *a time instant* t, *a pair of levels* $l_a, l_b \in L_A$, *a pair of elements* $x_a \in Ainstset(l_a, t)$ *and* $x_b \in Ainstset(l_b, t)$; $TReclassify(d, l_a, l_b, x_a, x_b, t)$ *updates the rollup functions in the dimension in the following way:*

- $\rho[t]_{l_a}^{l_b}(x_a) = x_b$;
- $\rho[t]_{l_i}^{l_j} = \rho[t^-]_{l_i}^{l_j}$, *for all other levels* $l_i, l_j \in L_A$.
- *The new dimension is snapshot consistent at time* t.

As in the non-temporal model, Temporal Reclassification is not always well-defined, and we will show this through an example.

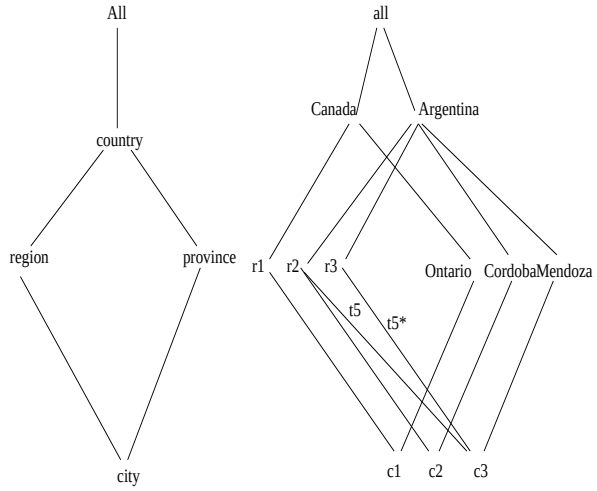

Figure 5.9: Temporal Reclassification

Example 32 *Let us suppose that in the dimension instance depicted in Figure 5.9(we used hypothetical city an region names for the sake of clarity) we want to reassign regions at time t_1, moving city c_2 to region r_1. This will obviously be not a valid operation, as it will not verify snapshot consistency at time t_1. The proposed reclassification would imply that c_2 belongs to Argentina and Canada at the same time. However, reclassification among cities and regions in the same country, will be valid. For instance, we could move c_3 to r_2, at time t_5, as it is shown in the figure.*

Level *Country* in Figure 5.9, is a *Confliciting Level* at time t_1 for the proposed reclassification(see Definition 13).

Lemma 5 (Definiteness of the Temporal Reclassify Operator) $\mathrm{TReclassify(d, l_a, l_b, x_a, x_b)}$ *is defined if and only if for every conflicting level l_k, $\rho^{*l_k}_{l_b}[t^-](x) = \rho^{*l_k}_{l_b}[t^-](x_b)$ holds, where $\rho^{l_b}_{l_a}[t^-](x_a) = x$.*

Proof Lemma 5 *The proof is exactly the same as the proof of Lemma 2, considering a snapshot at time t. Thus, we will not repeat the proof here.*

Operator 20 (Temporal Split) *Given a temporal dimension $d=((dname, L, \preceq\ \mu), TRUP)$, a time instant* t, *a level $l_a \in L_A(t^-)$, an element $x_a \in Ainstset(l_a, t^-)$, a list E of the form $\{x_{a1} \ldots x_{an}\}$, where $x_{ai} \in dom(l_a) \setminus Ainstset(l_a, t^-)$, another list P of the form $P = \{x_{a1}[l_1 : list_1 \ldots l_m : list_m]; \ldots; x_{an}[l_1 : list_1 \ldots l_m : list_m]\}$, where $l_i \preceq_t l_a, i = 1..m$, and* $\mathrm{list_i}$ *is a list of elements in* $\mathrm{Ainstset(l_i, t^-)}$, *of the form $(x_1, \ldots, x_k)$ s.t. $\rho[t]^{l_a}_{l_i}(x_i) = x_a$;* $\mathrm{TSplit(d, l_a, x_a, P, E, t)}$ *updates the rollup functions in the dimension in the following way:*

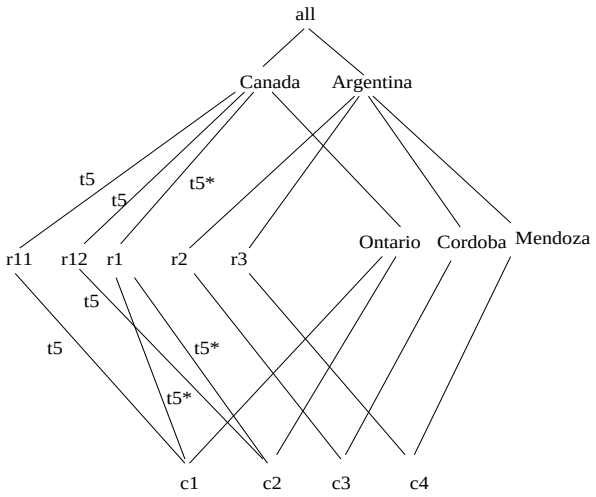

Figure 5.10: Temporal Split

- $\rho[t]_{l_a}^{l_i}(x_{aj}) = x_i, \quad where \quad x_{aj} \in E, x_i \in list_i \ s.t. \ x_{aj}[l_i : list_i] \in P;$
- $\rho[t]_{l_a}^{l_i}(x_{aj}) = x_i, \quad where \quad x_{aj}[l_i : list_i] \in P, x_{aj} \in E, RUP_{l_a}^{l_i}[t^-](x_a) = x_i;$
- $\rho[t]_{l_a}^{l_j}(x_a) = NIL, \forall l_j \quad s.t. \quad l_a \preceq_{t^-} l_j;$
- $\rho[t]_{l_i}^{l_j} = \rho[t^-]_{l_i}^{l_j}$, *for all other levels* $l_i, l_j \in L_A(t)$.

Example 33 *Suppose the schema of Figure 5.9, with the instance depicted in Figure 5.10. At time* t_5, *region* r_1 *is split into* r_{11} *and* r_{12}. *City* c_1 *is assigned to* r_{11} *and city* c_2 *to region* r_{12}. *Both new regions will still roll up to Canada, as* r_1 *did. The operation will be invoked as:* $TSplit(Geography, region, r_1, \{r_{11}, r_{12}\}, \{r_{11}[cityId : (c_1)]; r_{12}[cityId : (c_2)]\}, t_5)$. *Note that the user must assign the roll up functions corresponding to the new values* r_{11} *and* r_{12}.

The *Temporal Merge* operator performs the inverse of *TSplit*, i.e., it merges two instances of a dimension into a single one.

Operator 21 (Merge operator) *Given a temporal dimension* $d=((dname, L, \lambda, \preceq, \mu), TRUP)$, *a time instant* t, *a level* $l_a \in L_A(t^-)$, *an element* $x_N | x_N \in dom(l_a) \wedge x_N \notin Ainstset(l_a, t^-)$, *s.t. all the elements* $x_i \in X$ *roll up to the same element in every level* l *s.t.* $l_a \preceq_{t^-} l$; $Merge(d, l_a, X, x_N, t)$ *updates the rollup funcitions in the dimension in the following way:*

- $\rho[t]_{l_i}^{l_a}(x_i) = x_N, \quad where \quad \rho[t^-]_{l_i}^{l_a}(x_i) = x_i, x_i \in X;$
- $\rho[t]_{l_a}^{l_j}(x_N) = x_j, \quad where \quad \rho[\{t^-\}]_{l_a}^{l_j}(x_i) = x_j, \quad x_i \in X;$

- $\rho[t]^{l_j}_{l_a}(x_i) = NIL, \quad$ *where* $x_i \in X$;
- $\rho[t]^{l_j}_{l_i} = \rho[t^-]^{l_j}_{l_i}$, *for all other levels* $l_i, l_j \in L_A(t)$.
- *The new dimension remains snapshot consistent.*

Example 34 *Figure 5.11 shows another instance of the dimension* Geography. *Here, regions* r_2 *and* r_3 *were* merged *at* t_5 *into a new region* r_{23}. *All the cities which before* t_5 *rolled up to either* r_2 *or* r_3 *will now roll up to* r_{23}, *which will roll up to* Argentina. *The operation will be invoked as* $Merge(Geography, region, \{r_2, r_3\}, r_{23}, t_5)$, *and it can also be seen that it keeps the dimension in a consistent state.*

Operator 22 (Temporal Update) *Given a dimension* $d = ((dname, L, \lambda, \preceq, \mu), TRUP, \mu)$, *a level* $l_a \in L_A(t^-)$, *an element* $x_a \in Ainstset(l_a, t^-)$, *and an element* $x_n \notin Ainstset(l_a, t^-)$; $Update(d, l_a, x_a, x_n, t)$ *will update the dimension in the following way:*

- $\rho[t]^{l_j}_{l_a}(x_n) = x_j, \quad where \quad \rho[t^-]^{l_j}_{l_a}(x_a) = x_j$;
- $\rho[t]^{l_a}_{l_j}(x_j) = x_n, \quad where \quad \rho[t^-]^{l_a}_{l_j}(x_j) = x_a$;
- $\rho[t]^{l_j}_{l_a}(x_a) = NIL \quad \forall l_j \in L_A(t), \quad l_a \preceq_t l_j$;
- $\rho[t]^{l_a}_{l_j}(x_j) = NIL \quad \forall l_j \in L_A(t), \quad l_a \preceq_t l_j$, *and* $\rho[t]^{l_a}_{l_j}(x_j) = x_a$;
- $\rho[t]^{l_j}_{l_i} = \rho[t^-]^{l_j}_{l_i}$, *for all other levels* $l_i, l_j \in L_A(t)$.

5.4.5 Temporal Multidimensional Model Revisited

So far we have introduced a temporal version of the dimension update operators. These operators allow keeping track of the history of a dimension. However, in order to display the history of the elements in a dimension level, for instance, of region r_1 in example 33, we need to add two predicates to the data model, denoted $split(x, y, L, t)$, and $merged(x, y, L, t)$, with the following meanings: $split(x, y, L, t)$ is *true* if the element x, in level L was split at time t, and y is one of the elements resulting from this splitting; (b) $merged(x, y, L, t)$ is *true* if element x in level L was merged into node y at time t. Predicates *split* and *merged* are *event* predicates, in temporal database sense.

In the next chapter we will present a temporal query language. We will see that the expressive power of this language can be expanded making use of the predicates described above.

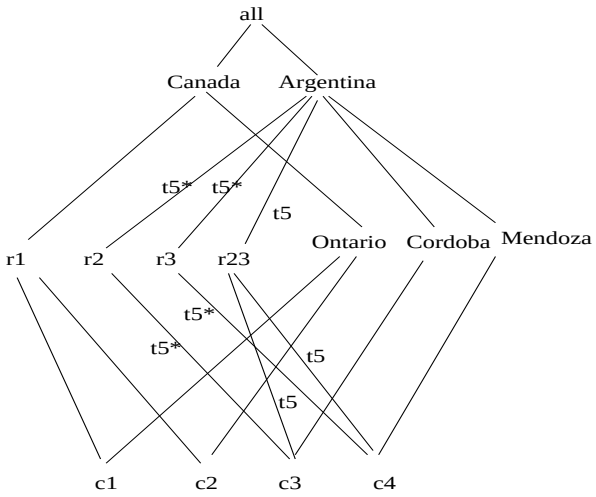

Figure 5.11: Temporal Merge

5.5 Summary

In this chapter we introduced the *Temporal Multidimensional Model*, and the temporal extension of the dimension update operators introduced in Chapter 2. In the next chapter we will present a temporal query language supporting the model and discuss its implementation.

Chapter 6

TOLAP : Temporal OLAP Query Language

We argued in previous chapters that usually in an OLAP environment, queries require the computation of aggregates over base fact tables. Moreover, in Chapter 5 we introduced a model accounting for temporal dimensions, i.e. dimensions that evolve across time, allowing to keep track of the dimension's history. We denoted this dimensions *temporal*. We will show in this chapter that typical OLAP queries involving temporal dimensions admit different interpretations, which should be considered in order to give the user the correct answers to queries. Further, we introduce a query language called TOLAP supporting the model presented in Chapter 5.

6.1 Introduction

There are many real-life situations in which maintaining historical data is needed in order to get correct answers to OLAP queries. Consider for instance an NBA (professional basketball) data warehouse where the fact table *Points* has dimensions *Player* and *Time*, and a measure, *Points scored*. The *Player* dimension is structured by grouping players into teams called *Franchises*. Suppose a user wants to know the total number of points scored by the players of the Portland Blazers. This query could be interpreted in two different ways: the user could be asking for the sum of total points ever scored by all players who are currently on the Blazers, or for the sum of the points scored by these same players while playing for the Blazers. For instance, the points scored by Damon Stoudamire (who is currently on the Blazers) while playing for the Toronto Raptors in

the 1998-99 season should only be added under the first interpretation. Query languages provided by most standard commercial OLAP systems will not be able to distinguish one interpretation from the other. The reason is that these systems just record the last value of dimensional attributes giving no access to their historic values. The Temporal Multidimensional Model introduced in Chapter 5 allows defining powerful languages which can express adequately a query like the one above.

6.1.1 Our Proposal

In this chapter we present a temporal query language supporting the Temporal Multidimensional Model introduced in Chapter 5. We call this language *TOLAP* (standing for *Temporal OLAP*). *TOLAP* combines some of the features of temporal query languages like TSQL2 or SQL/TP [Sno95, Tom97] with some of the high-order features of languages like HiLog or SchemaLog [CKW89, LSS97], in the OLAP setting. We introduce *TOLAP* by means of examples, formally define its syntax and semantics, and discuss its expressive power We also show that *TOLAP* can be easily extended in order to allow queries in which the history of the dimension's elements is taken into account.

Let us show how the first interpretation of the NBA query of Section 6.1 can be expressed in TOLAP:

$$\texttt{Q(x,SUM(p))} \longleftarrow \texttt{Points(x,p,t), x} \xrightarrow{Now} \texttt{franchise:'Blazers'.}$$

This means: for each player `x`, add up all the points scored by `x`, where `x` is such that currently belongs ("rolls up") to the Blazer franchise. The query for the second interpretation will read:

$$\texttt{Q(x,SUM(p))} \longleftarrow \texttt{Points(x,p,t), x} \xrightarrow{t} \texttt{franchise:'Blazers'.}$$

Here we are asking to add the points scored by `x` at time t, if `x` at that time was playing for the Blazers.

Descriptive attributes make queries like *"total number of points scored by Stoudamire while playing for the Toronto Raptors"* easy to express:

```
Q(SUM(p))  ⟵  Points(x,p,t), x ⟶ᵗ franchise:'Raptors', x.name=ᵗ'Stoudamire'.
```

Note that the queries above operate at a high level of abstraction, without requiring low-level knowledge about the database design that underlies the dimensional model.

6.1.2 Do We Need a Temporal OLAP Language?

One might argue, at first sight, that a generic temporal query language like TSQL2 [Sno95] could be used instead of defining a special-purpose one like *TOLAP*. There are two main reasons supporting the idea of introducing a new language. First, a language designed specifically for the multidimensional model makes typical OLAP queries much more concise and elegant. In a generic language, queries would have to be laboriously encoded using detailed knowledge of the low-level relational structures used to encode the dimensional data. Second, the best-known temporal query languages such as TSQL2, support only a minimal level of schema versioning ([Sno95] p.29).

Another alternative would have been adding temporal features to other languages with schema management features, such as HiLog [CKW89] or SchemaLog [LSS97]. Again, using a language specifically designed for OLAP yields much simpler syntax and semantics, and just the high-order features that are needed to support schema evolution.

The remainder of this chapter is organized as follows. In Section 6.2 we motivate the need for a temporal query language for OLAP. In Section 6.3 we introduce *TOLAP*. The syntax and semantics of the language are detailed in Section 6.4. In Section 6.5 we discuss *TOLAP*'s expressive power and a possible extension of the language. We conclude in Section 6.6.

6.2 Motivating Example

The following example will show that a temporal OLAP query language is needed in order to capture the particular characteristics of OLAP queries expressed over a set of temporal dimensions. We will use this example when introducing our query language in Section 6.3.

Example 35 *Let us consider again a retail data warehouse, with a set of temporal dimensions* $\mathbf{D} = \{Product, Store\}$*, and a base fact table with schema* $(Sales, f, sales, day)$*. At this time we will assume that no schema update occurred (this will be studied in the next sections). Thus,* f *maps each instant to* $\{itemId, storeId\}$*, the bottom levels of the dimensions in D. Dimen-*

sions Store *and* Product *are depicted in Figures 6.1 and 6.2, respectively. Notice that in* Product, *item* i_1 *has type* t_1 *until day* d_4, *and type* t_2 *since then. For the* Time *dimension we have:* $\rho_{day}^{week} = \{d_1 \rightarrow w_1, d_2 \rightarrow w_1, d_3 \rightarrow w_2, d_4 \rightarrow w_2, d_5 \rightarrow w_2\}$*(and the rollups from week to All). Finally, we have the following instance for the* Sales *fact table (*day *is displayed for the sake of clarity, but could have been omitted, like in TSQL2):*

itemId	storeId	day	sales
i_1	s_1	d_1	600
i_2	s_2	d_1	100
i_2	s_1	d_2	100
i_3	s_2	d_2	100
i_3	s_3	d_3	100
i_3	s_4	d_4	100
i_1	s_1	d_5	100

Let us now suppose the query: " list the weekly total sum of sales, by city and item type". As in the NBA example of Section 6.1, two interpretations could be given to this query. The most usual one would expect to get the sum of sales considering the type an item had when it was sold. In this case, for instance, item i_1 *would contribute to the aggregation in the following way: the first three tuples, with a total of 800, will add to the group* $\{t_1, c_1, w_1\}$, *while the last one will contribute to* $\{t_2, c_1, w_2\}$. *The result will be given by the following table:*

itemType	city	week	sales
t_1	c_1	w_1	800
t_2	c_1	w_1	100
t_2	c_2	w_2	200
t_2	c_1	w_2	100

The second interpretation, the **only** *one supported by non-temporal systems, would ask for the sum of the sales, considering that each sold item has the* current *type, regardless of the time the sale occurred. The result a user would get under this interpretation is given by the table below, and was computed in the following way: the rollup function for every occurrence of item* i_1 *is set to:* $\rho_{itemId}^{itemType}(i_1) = t_2$. *Thus, all the* i_1 *tuples will contribute to type* t_2. *For instance, the first tuple will*

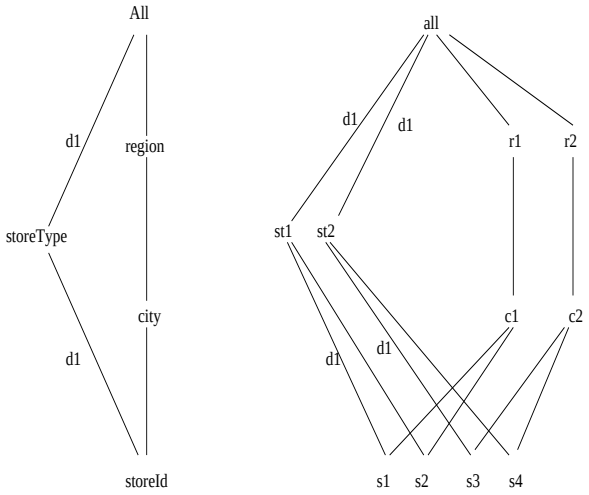

Figure 6.1: Dimension *Store* for the running example

now contribute to the group $\{t_2, c_1, w_1\}$. *In the next Section we will present a language that lets the user distinguish between the two interpretations. The following table shows the result the user would obtain under the second interpretation.*

itemType	city	week	sales
t_1	c_1	w_1	200
t_2	c_1	w_1	700
t_2	c_2	w_2	200
t_2	c_1	w_2	100

6.3 *TOLAP*: A Temporal Multidimensional Query Language

Temporal OLAP queries require a language which can account for their particular characteristics. Thus, in this section we introduce *TOLAP* (Temporal OLAP), a multidimensional query language. We will first present *TOLAP* by means of examples, and then define its syntax and semantics. In Section 7.4 we will apply *TOLAP* to the case study of Chapter 3.

6.3.1 TOLAP By Example

Let us consider again the set of dimensions $\mathbf{D} = \{Product, Store\}$ and the base fact table *Sales*, from Example 35. In the *Product* dimension, $\mu = day$. Also assume there is a fact table

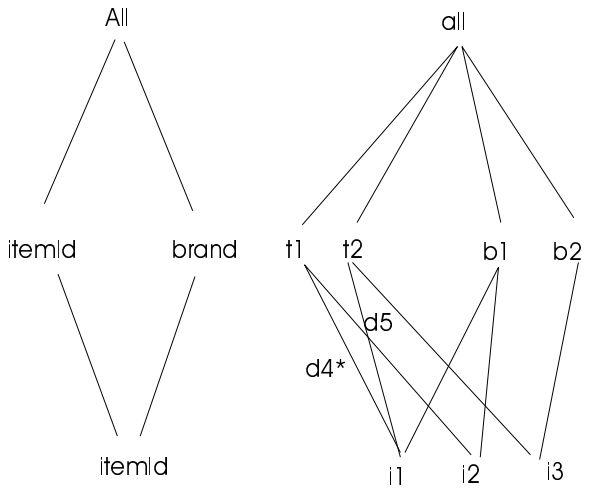

Figure 6.2: Dimension *Product* for the running example

$(Price, f_D, price, \mu : month)$, containing the price of each item each month($f_D(t) = \{itemId\}$ for each t).

Simple Queries

We begin with queries not involving aggregates.

Example 36 *A query returning the sales for stores in Buenos Aires, on a daily basis, will be expressed in* TOLAP *as:*

$$\texttt{BASales(p,s,m,t)} \longleftarrow \texttt{Sales(p,s,m,t), s} \xrightarrow{t_1} \texttt{city:'BA', t} \longrightarrow \texttt{month:}t_1.$$

In TOLAP*, the query above returns the tuples in* Sales *such that* s *rolled up to* 'BA' *at the time of the sale, where* s *represents an element in the instance set of the lowest level of the dimension* Store*. The atom* $\texttt{t} \longrightarrow \texttt{month:}t_1$ *explicitly performs the conversion between the granularities of the fact table(day) and the dimension table* Store*(month). Our actual implementation of TOLAP hides this granularity management from the user. The query is expressed in a* point-based fashion *(see Section 6.4.2 for details), and is interpreted as follows: a tuple in* Sales *will be in the result if a product* p *was sold in store* s *on a day* t *belonging to month* t_1*, if* s *was settled in Buenos Aires on that month.*

We assume a fixed order of the attributes in the base fact tables. For instance, in the base fact table *Sales,* the first position from the left will always correspond to dimension *Product.*

Example 37 *The next query asks for the dollar value of the daily sales for each store and item.*

DailySales(p,s, d*r,t) $\longleftarrow$ Sales(p,s,d,t), Price(p,r,m), t $\longrightarrow$ month:m.

Here, we used a scalar function "". This, along with other scalar functions, will be assumed as predefined interpreted functions. The atom t $\longrightarrow$ month:m homogenizes granularities between the two fact tables in the body of the rule.*

Queries With Aggregates

In order to address queries involving aggregation, we adapt non-recursive datalog with aggregate functions [CM90], which, in turn, was based on the approach of Klug's relational calculus with aggregates [Klu82].

Example 38 *Consider the query : "list the total sales per item and region," where we want aggregates to be computed using temporally consistent values (i.e., a sale in a given store is credited to the region that corresponded to that store at the time of the sale).*

IR(it,re,SUM(m)) $\longleftarrow$ Sales(it,st,m,d), st $\stackrel{mo}{\longrightarrow}$ region:re,
d $\longrightarrow$ month:mo.

Note that although in Example 38 we made explicit the rollup between the time granularity of the *Sales* and *Store* dimensions (i.e. *day* and *month*), this could be easily avoided, allowing a limited form of "schema independence", as in Schemalog or SchemaSQL [LSS93, LSS97]. Later examples show the use of variables that range over level names, pushing this independence farther.

In *TOLAP,* time management can be hidden from the user. Taking advantage of this feature, the query of Example 38 will read:

IR(it,re,SUM(m)) $\longleftarrow$ Sales(it,st,m,d), st $\stackrel{d}{\longrightarrow}$ region:re.

In this query, the user does not express the rollup from day to month. In the former one, the rollup from *day* to *month* had been made explicit with the atom d $\longrightarrow$ month:mo,, and the use of the variable mo in the rollup expression st $\stackrel{mo}{\longrightarrow}$ region:re. *TOLAP* automatically checks granularities and performs the necessary conversions. This approach will be followed in the remainder of this chapter.

Example 39 *We now introduce descriptive attributes of dimension levels. Suppose we want the total sales by store and brand, for stores with more than ninety employees. Assume that the level* storeId *is described by an attribute* nbrEmp *(number of employees).*

$$\texttt{SB(br,st,SUM(m))} \longleftarrow \texttt{Sales(i,s,m,t), i} \xrightarrow{t} \texttt{brand:br,}$$
$$\texttt{s} \xrightarrow{t} \texttt{storeId:st, s.nbrEmp} \geq 90.$$

Metaqueries

TOLAP also allows querying the system's metadata. Thus, *TOLAP* supports queries with no fact table in the body of the rules. Some examples of these kinds of queries are:

- *"Give me the time instants at which store s_1 belonged to the Southern region"*, expressed as:

  $$\texttt{StoreTime(t)} \longleftarrow \texttt{Store:storeId:`}s_1\texttt{'} \xrightarrow{t} \texttt{region:`Southern'.}$$

  Note that we must specify the name of the dimension in the atom Store:storeId:'s_1', because there is no fact table in the body of the rule to which s could be bound.

- *"List the months when "Southern" was not a valid region".*

  $$\texttt{noSouth(t)} \longleftarrow \neg\texttt{Store:region:`Southern'} \xrightarrow{t} \texttt{X:x.}$$

  In the example above, variable X ranges over level names in the *Store* dimension. Variable x is bound by the values in the levels of the dimension. The domain of t is the lifespan of the dimension referenced in the metaquery.

- *"Were products categorized by brands two years ago?"* .

  $$\texttt{ProdBrand()} \longleftarrow \texttt{Product:X:x} \xrightarrow{1/1/98} \texttt{brand:y.}$$

  Here again X ranges over level names. The expression above means that if any element, in any level in the *Product* dimension rolled up to an element in level *brand* at the required date, the answer to the query will be '*yes*'.

- *"How were customers classified three years ago?".*

```
CustCat(cust,X,x) ⟵ Customer:customerId:cust -1/1/97-> X:x.
```

6.3.2 Data Warehouse Evolution in *TOLAP*

In Section 5.3 we showed that our model supports evolution of the schema over time (in temporal database terminology this is called *schema versioning* or *schema evolution*). For instance, suppose in our running example that the bottom level of the *Store* dimension in Figure 6.1 was initially *city*, and that, at time d_5, *storeId* was inserted below it. Also assume that the fact table depicted below was in effect *before* d_5.

itemId	city	sales	day
i_1	c_1	100	d_1
i_2	c_2	100	d_2
i_1	c_3	100	d_3
i_2	c_1	100	d_1

After the update, the following sales occurred (notice the new structure of the fact table):

itemId	storeId	sales	day
i_1	s_4	100	d_6
i_2	s_2	100	d_6
i_1	s_3	100	d_7
i_2	s_1	100	d_7

In *TOLAP*, an element not defined at a given instant, will not contribute to the result. For instance, given the query *"list the total sum of sales by brand and storeId"*, we have:

```
SB(br,st,SUM(m)) ⟵ Sales(i,s,m,t), i -t->brand:br,
                   s -t->storeId:st.
```

The expression `s`$\xrightarrow{t}$`storeId:st` means that if an element in any level, which was once a component of a base fact table, rolled up to level *storeId* at time t, it contributes to the aggregation. Thus, the sales made before the month corresponding to d_5 will not contribute to the aggregation in the head (condition `s`$\xrightarrow{t}$ `storeId:st` will not be satisfied). Analogously, a query like *"total*

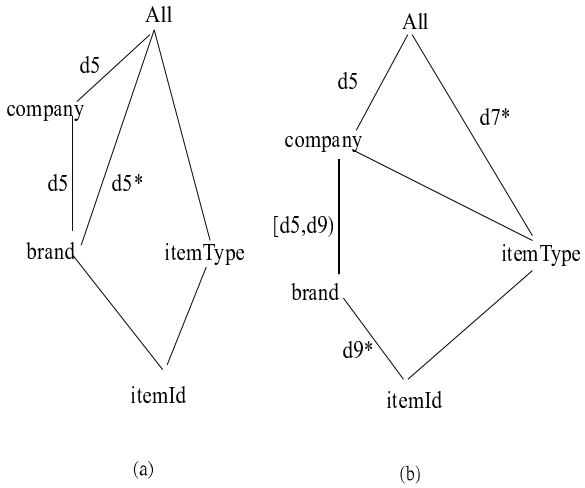

Figure 6.3: Data warehouse evolution

sales by store and itemId" would return exactly the instance of the second fact table above. This query is expressed:

$$\texttt{StIt(it,st,SUM(m))} \longleftarrow \texttt{Sales(i,s,m,t), i} \xrightarrow{t} \texttt{itemId:it, s} \xrightarrow{t} \texttt{storeId:st.}$$

Figure 6.3(a) shows that at time d_5, level *company* was added above level *brand* (a *generalization* from *brand* to *company*) in dimension *Product.* Given the query : *"total sales by company and region"*, the sales occurring before d_5 will not contribute to the aggregation, as *company* was not a level of the dimension at that time. The query reads in *TOLAP*:

$$\texttt{CR(c,reg,SUM(m))} \longleftarrow \texttt{Sales(i,s,m,t), i} \xrightarrow{t} \texttt{company:c, s} \xrightarrow{t} \texttt{region:reg.}$$

Finally, suppose that at time d_9, level *brand* is deleted from the dimension *Product.* The remaining levels are *itemId*,(bottom level), *itemType*, *company*, and *All*. The query *"total sales by brand and region"* is expressed in *TOLAP* as:

$$\texttt{BR(br,reg,SUM(m))} \longleftarrow \texttt{Sales(i,s,m,t), i} \xrightarrow{t} \texttt{brand:br, s} \xrightarrow{t} \texttt{region:reg.}$$

Any sale occurred after d_9 will not be considered, as *brand* is not a level of the dimension anymore.

6.3.3 *TOLAP* Programs

A *TOLAP* program allows to compute the result of a rule and use it as a predicate in the body of another rule. The predicate's name is the name of the predicate in the head of the rule which computes it.

For instance, let us suppose the following query: "List the total sales by brand, for those brands that sold more than one hundred thousand dollars in Buenos Aires". We want to answer this query precomputing a view holding the total sales in Buenos Aires, by brand.

```
BASales(c,SUM(m)) ⟵ Sales(a,s,m,t), a -t->brand:c,
                    s -t->city:x, x.name=''Buenos Aires''.
```

Then, we compute the query, using predicate *BASales*.

```
Q(c,SUM(m)) ⟵ Sales(a,s,m,t), BASales(c,q), a -t->brand:c, q ≥ 100000.
```

For each brand in dimension *Product* matching a brand in BASales, in the second rule, if the amount sold was greater than one hundred thousand, the sales contributes to the aggregation.

6.4 TOLAP Syntax and Semantics

6.4.1 Syntax

In this section we will formally define the syntax of a *TOLAP* rule. We will first give some definitions which will be used below, and then formalize the concepts introduced in Section 6.3.

Preliminary Definitions

Given a set T, and a discrete linear order $<$, with no endpoints, we define a *point based temporal domain* as the structure $T_P = (T, <)$. Analogously, given $T_P = (T, <)$, we define the set $I(T) = \{(a, b) | a \leq b, a, b \in T \cup \{-\infty, +\infty\}\}$. Let us denote by θ the set of the usual interval comparison operators. Then, $T_I = (I(T), \theta)$ is an *Interval-based Temporal Domain* corresponding to T_P. [Tom97]. The rollup functions in *TOLAP* will be defined over T_P.

Atoms, Terms, Rules and Programs

Assume $\mathbf{B_s}(\mathbf{D_s}, \mathbf{F_s})$ and $I(\mathbf{B})$ represent a multidimensional database schema and instance, respectively, as defined in Section 5.3. Let $\mathbf{V_L}$ and $\mathbf{V_D}$ be a set of level and data variables, respectively. We have also the sets $\mathbf{C_L}$ and $\mathbf{C_D}$ of level and data constants, respectively. Let $\mathbf{P}$ be a set of predicate names, and $\mathbf{F_{Ag}}$ a set of aggregate function names.

Definition 23 (Terms)

- *A* data term *is either a variable in* $\mathbf{V_D}$ *or a constant in* $\mathbf{C_D}$;
- *a* rollup term *is an expression of the form* `d:X:x, X:x` *or* `x,` *where* `X` *is a level name variable in* $\mathbf{V_L}$ *or constant in* $\mathbf{C_L}$, `x` *is a data term, and* `d` *is a constant in* $\mathbf{C_L}$;
- *a* descriptive term *is an expression of the form* `x.a` *where* `x` *and* `a` *are data terms;*
- *an* aggregate term *is an expression of the form* `f(d)` *such that* `f` *is a function name in* $\mathbf{F_{Ag}}$ *and* `d` *is a data term.*

A term *is a* data, rollup, descriptive *or* aggregate *term.*

Definition 24 (Atoms)

- *A* fact atom *is an expression of the form* $\mathtt{F}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{M}, \mathtt{t})$, *where* `F` *is a fact table name in* $\mathbf{F_s}$, *and* $\mathtt{X_1}, \ldots, \mathtt{X_n}$, `M` *and* `t` *are data terms;*
- *a* rollup atom *is an expression of the form* $\mathtt{X} \xrightarrow{t} \mathtt{Y}$, *or* $\mathtt{X} \longrightarrow \mathtt{Y}$, *where* `X` *and* `Y` *are rollup terms, and t is a data term;*
- *a* descriptive atom *is an expression of the form* $\mathtt{x} \stackrel{t}{=} \mathtt{y}$, *where* `x` *is a descriptive term, and* `y` *and t are data terms;*
- *an* aggregate atom *is of the form* $\mathtt{Q}(\mathtt{R}, \ldots, \mathtt{Z})$ *s.t.* $\mathtt{Q} \in \mathbf{P}$, *and* $\mathtt{R}, \ldots, \mathtt{Z}$ *are data terms s.t. at least one is an aggregate term;*
- *a* constraint atom *is an expression* $\mathtt{t_1}\ \theta\ \mathtt{t_2}$, *where* $\mathtt{t_1}$ *and* $\mathtt{t_2}$ *are data terms, and* θ *is one of* $\{<, =\}$;
- *if* $\mathtt{g} : \mathtt{N} \times .. \times \mathtt{N} \rightarrow \mathtt{N}$ *is a scalar function,* $\mathtt{g}(\mathtt{n_1}, \ldots \mathtt{n_m})$, *where* $\mathtt{n_i}$ *are data terms, is a* scalar atom;

- *an* intensional(extensional) predicate atom *is an expression of the form* `p (X,..,Z)` *where* `X,Y,Z` *are data terms, and* `p` *is an intensional(extensional) predicate name in* $\mathbf{P}$. *In what follows, we will refer to this kinds of atoms simply as* predicate *atoms when possible.*

An atom *is a fact, rollup, descriptive, aggregate, constraint, scalar or predicate atom. An expression* $\neg \mathtt{t}_1$, *where* $\mathtt{t}_1$ *is an atom, is a* negated atom.

Definition 25 *(TOLAP Rules) A* TOLAP rule *is a formula of the form* $\mathtt{A} \longleftarrow \mathtt{A}_1, \mathtt{A}_2, ...\mathtt{A}_n$, *where A is an intensional(possibly aggregate) positive atom, and* $\mathtt{A}_i$, $\mathtt{i} = 1 \ldots \mathtt{n}$ *are non-aggregate atoms. A TOLAP rule* Γ *satisfies the following conditions:*

- *if a variable appears in the head of the rule, it must also appear in its body;*
- *for every level/data variable v, all the rollup terms in which v appears are associated to the same dimension; moreover, all the variables in a rollup atom belong to the same dimension;*
- *if there is an aggregate atom* $\mathtt{Q}(\mathtt{a}_1, \ldots, \mathtt{a}_n)$ *in the head of the rule, for all atoms in the body, of the form* $\mathtt{d:X:x} \xrightarrow{t} \mathtt{y} : \mathtt{a}_i$, $\mathtt{x} \xrightarrow{t} \mathtt{y} : \mathtt{a}_i$, `y` *is a constant data term (thus, aggregation is performed over constant level names);*
- *every variable* `x` *in a rollup atom of the form* $\mathtt{x} \xrightarrow{t} \mathtt{y:a,}$ *must appear in a fact atom in the body of the rule;*
- *if* `x.a` *is in the body of* Γ, *at least one rollup term in the body is of the form* `d:X:x`, `X:x` *or* `x;`
- *every variable appearing in a predicate atom must appear in a fact or rollup atom in the body of the rule;*
- *in a rollup term of the form* $\mathtt{d:X:x} \xrightarrow{t} \mathtt{y:a,}$ `d` *is a constant data term;*
- *if there is no fact atom in the body of* Γ, *all the rollup atoms must be of the form* $\mathtt{d:X:x} \xrightarrow{t} \mathtt{y:a;}$
- *In a negated* fact *atom, at least one of its terms must be a constant in* $\mathbf{C_D}$;
- *if* $t_1 \theta t_2$ *is a constraint atom,* t_1 *and* t_2 *are both variables in a fact, rollup or descriptive atom in the body, or one of them is a constant in* $\mathbf{C_D}$ *and the other is a variable in a fact, rollup or descriptive atom.*

- *if the head of a rule is of the form* $\mathtt{Q}(\mathtt{a}_1, \ldots, \mathtt{a}_n)$, $\mathtt{Q}$ *cannot appear in the body of the same rule.*

  From the rules above, it follows that there is a function, call it `dim`, *that maps each data variable* `x` *to a unique dimension* `dim(x)`, *and each level variable* X *to a unique dimension* `dim(X)`. *Furthermore, there is a function* `level`, *that maps each instant* `t` *to a unique level* `level(x,t)` *of the dimension* `dim(x)`, *and to a unique level* `level(X,t)` *of the dimension* `dim(X)`.

- *The granularity of a fact table* F *must be finer than the finest granularity of any of its component dimensions involved in a rollup atom in the body of the rule.*

  This rule prevents a fact from being counted more than once. For example, consider the query:

  `Q(is,rs,SUM(m))` $\longleftarrow$ `F(i,s,m,t), s` $\xrightarrow{t}$ `d:rs.`

  Assume that the fact table granularity is "year", and the granularity of `dim(s)` *is "day". A fact in an instance of* `F` *could be of the form* $< i_1, s_1, 100, 1998 >$ *. The dimension may have a pair of tuples* $< s_1, rs_1, 1-1-1998, 5-5-1998 >$ *and* $< s_1, rs_2, 5-6-1998, 12-31-1998 >$ *. The fact would be counted twice. Thus, we do not admit this case to occur.*

Definition 26 (Mutual Recursion) *Given a set* $\mathcal{R}$ *of TOLAP rules, a* precedence graph G *is built as follows: for each aggregate, predicate or fact atom* P, *there is a node named* P *in* G. *If* P *and* Q *are nodes in* G, *add an edge from* P *to* Q *if there is a rule* Γ *in* $\mathcal{R}$ *such that* P *and* Q *occur in the body and the head of* Γ, *respectively. Following Abiteboul et al [AHV95] we say that two aggregate, fact or predicate atoms* R *and* S *are mutually recursive if* R *and* S *participate in the same cycle in* G. *We do not consider the case* R=S *because a TOLAP rule is non-recursive by definition.*

Definition 27 (*TOLAP* Programs) *A finite set of* TOLAP rules *which does not contain mutually recursive atoms is called a* TOLAP Program.

6.4.2 Semantics

We will use point-based semantics [Tom95] for the rollup functions. This means, for instance, that in a dimension such as *Store* of our running example, a value for a rollup, say `storeId:s:'i1'` $\xrightarrow{mo}$ `city:c:'c1'`, exists for each month in its validity interval.

Let us assume that for each dimension instance we have a pair of relations, call them $\mathcal{R}_\mathcal{D}$ and $\mathcal{D}_\mathcal{D}$, representing the sets *TRUP* and *TDESC* of Definition 18, respectively. The multidimensional database is defined over three different domains: $\mathcal{D}$, N, T_P, where variables ranging over $\mathcal{D}$ belong to an uninterpreted sort, the ones ranging over N belong to an interpreted sort (numeric), and the temporal variables range over T_P, defined in 6.4.1. For each dimension d, we will consider that T_P is limited to the lifespan of d.

A *valuation* θ for a *TOLAP* rule Γ, is a tuple (θ_s, θ_I), where θ_s is called a *schema valuation,* and θ_I is an *instance valuation.* Valuation θ_s maps the level and attribute variables in Γ to level and attribute names in $\mathbf{B_s}(\mathbf{D_s}, \mathbf{F_s})$, while θ_I maps domain variables to values in $I(\mathbf{B})$.

Definition 28 (Valuations) *A* schema valuation *for a rule* Γ, *denoted* $\theta_s(\Gamma)$ *maps level and attribute variables in the atoms of* Γ *as follows:*

- *given a rollup atom of the form* $\texttt{d:X:x} \xrightarrow{t} \texttt{Y:y}$, θ_s *maps* d *to a dimension name in* $\mathbf{D}_s$, t *to a value* $w \in T_P$, *and* X *and* Y *to a pair of values* (v, u) *s.t.* $v \preceq^*_w u$ *holds in* d;
- *if the rollup atom is of the form* $\texttt{X:x} \xrightarrow{t} \texttt{Y:y}$, θ_s *maps* t *to a value* $w \in T_P$, *and* X *and* Y *to a pair of values* v, u *s.t.* $v \preceq^*_w u$ *holds in* $\texttt{dim(X)} \in \mathbf{D}_s$;
- *if the rollup atom is of the form* $\texttt{x} \xrightarrow{t} \texttt{Y:y}$, θ_s *maps* t *to a value* $w \in T_P$, *and* Y *to a value* u *s.t.* $\texttt{level(x,w)} \preceq^*_w u$ *holds in* $\texttt{dim(x)} \in \mathbf{D}_s$;
- *for a rollup atom of the form* $\texttt{x} \longrightarrow \texttt{Y:y}$, θ_s *maps* Y *to a dimension level* u *in* dim(Y), *s.t.* $l_{inf} \preceq^* u$ *holds in* dim(X);
- *given a descriptive atom of the form* $\texttt{x.A} \stackrel{t}{=} \texttt{y}$, θ_s *maps* t *to a value* $w \in T_P$, *and* A *to an attribute name* $u \in \mathbf{A}$, *s.t.* $u \gg_t \texttt{level(x,w)}$ *in* $\texttt{dim(x)} \in \mathbf{D}_s$;

Given a rule schema valuation $\theta_s(\Gamma)$ *for a rule* Γ, *an* instance valuation *is a function* θ_I *s.t.*

- θ_I *maps the domain variables* x *and* y *in the rollup atoms defined above to values in* $\mathcal{R}_\mathcal{D}$ *over levels defined by* θ_s;
- θ_I *maps variable* x *in the descriptive atom* $\texttt{x.A} \stackrel{t}{=} \texttt{y}$ *to values in* $\mathcal{D}_\mathcal{D}$, *over levels defined by* θ_s, *and* y *to a value in* $\mathcal{D}$ *or* N;
- θ_I *maps a fact atom* $F(x_1, .., x_n, M, t)$ *as follows: the rightmost term* t *in* F *is mapped to a value* $w \in T_P$; *each domain variable* x_i *in* F *to a value in* $dom(\texttt{level}(\texttt{x}_\texttt{i}, \texttt{w}))$; *and the data term* M *in* F *to a value in* N *(*M *is the* measure *of the fact table).*

- *a* constraint atom $x \ \{<,=\} \ y$ *evaluates to* true *whenever* $\theta_I(x) \ \{<,=\} \ \theta_I(y)$ *is true. Recall that every variable in a constraint atom must appear in a rollup or fact atom in the body. Thus, either* θ_I *maps* x *and* y *in the way described above, or they are mapped to a value in* $\mathcal{D}$, N *or* T_P.

- *A* negated rollup atom *is evaluated using the Close World Assumption. Thus,* $\neg(\ \mathtt{x} \xrightarrow{t} \mathtt{Y:y})$ *is true if, given a valuation* θ *s.t.* $\theta(x) = u$, $\theta(t) = w$, $\theta(Y) = l$, *and* $\theta(y) = v$, *then there does not exist a rollup in* `dim(x)` *s.t.* $\rho^{l}_{\mathtt{level(x,w)}}[w](u) = v$. *Recall that every variable must be in a positive atom in the body of the rule.*

- *A* negated descriptive atom $\neg\mathtt{x.A} \stackrel{t}{=} \mathtt{y}$ *is treated as explained above for negated rollup atoms. Thus,* $\neg\mathtt{x.A} \stackrel{t}{=} \mathtt{y}$ *is true if, given a valuation* θ *s.t.* $\theta(x) = u$, $\theta(t) = w$, $\theta(A) = a$, *and* $\theta(y) = v$, *then it does not exist a description in* `dim(x)` *such that* $\xi[t]^{a}_{\mathtt{level(x,w)}}(u) = v$.

- *A* negated constraint atom *is evaluated in the standard way (i.e.* $\theta(\neg\mathtt{x} = \mathtt{b}) = \theta(\mathtt{x} <> \mathtt{b})$*)* .

- *A* negated fact atom $\neg F(x, y, \ldots, \text{“}a\text{”}, m, t)$ *is true if for a valuation* $\theta_I(x) = u$, $\theta_I(y) = v, ...,$ *the tuple* $< u, v, \ldots, \text{“}a\text{”}, ... >$ *is not in* F.

- predicate atoms *are valuated as in standard datalog [AHV95].*

Let AGG be the set of aggregate functions, with extension $AGG = \{MIN, MAX, COUNT, SUM, AVG\}$, and r a relation. The *aggregate operation* [CM90] $\gamma_{f_{A(X)}}(r)$ is the relation

$\gamma_{f_{A(X)}}(r) = \{t : t \text{ is an XA-tuple}, t[X] \in \pi_X(r), t[A] = f_A(\sigma_{X=t[X]}(r))\}$,

over XA, s.t. $XA \in schema(r), f \in AGG$, and $f_A(r)$ denotes the aggregation of the values in $t[A], t \in r$, using f.

Thus, we can now define the semantics of a $TOLAP$ rule Γ of the form

$Q(a_1, a_2, \ldots, a_n, AGG(m)) \longleftarrow A_1, \ldots, A_m$

as follows:

For each level or data variable v_i in the body of Γ, and for a valuation θ of the variables in the rule's body, we have:

$r_\Gamma = \{< \theta(v_1), \ldots, \theta(v_n) > | \theta \text{ is a valuation of } \Gamma\}$.

Then $Q = \gamma_{AGG_m(a_1,\ldots,a_n)}(r_\Gamma)$.

6.4.3 Discussing Safety

Queries expressed as *TOLAP* rules studied in the previous subsections always lead to finite answers. This follows from the syntactic restrictions and from the semantics defined in subsection 6.4.2. The existence of functions `dim(x)` and `level(x,t)` makes every variable in a rule *bounded* by the active domain of some level in a dimension. This is analogous to the concept of *range restricted variable* [AHV95]. Thus, for instance, if a negated rollup atom appears in the body, safety is always granted. The same occurs with negated fact atoms. Also, limiting the temporal domain to the lifespan of the dimensions avoids infinite query answers, like in the second metaquery in Section 6.3.1. Thus, we conclude that *TOLAP* rules are *safe.* We apply this conclusion in Section 7.2, where we translate a *TOLAP* rule to an SQL query.

6.5 Expressive Power

In this section we informally discuss *TOLAP*'s expressive power. We also define an extension to *TOLAP* which will allow expressing queries not supported by the syntax and semantics defined so far.

6.5.1 What Can Be Expressed in *TOLAP*?

Intuitively, it is not hard to see that *TOLAP* has at least the power of first-order query languages with aggregation. However, in a sense it goes beyond this class. Note that in the temporal multidimensional data model, only the direct rollups are stored. Thus, to evaluate a rollup atom like $\texttt{d:X:x} \xrightarrow{t} \texttt{Y:y}$, we need to compute the transitive closure of the rollup functions in dimension `d`. It is a well-known fact that this cannot be done in first-order, even after adding aggregate functions [LW97]. However, as long as the dimension schema is fixed, this computation can be done in first order, because for a fixed schema, the number of joins needed to transitively close the rollup functions is known in advance.

Observe that not only the structure of a dimension is subject to updates. There are common real-life situations in which an instance of a dimension may be modified in a non-trivial fashion. For example, splits, mergings or reclassification can occur.

Assume a "Geography" dimension is defined in the data warehouse we are using as our running example, with levels *city* and *region* , and the following query is posed: *"total sales per item and*

region, using only the currently existing regions". In *TOLAP* we would write:

$\texttt{SB(p,r,SUM(m))} \longleftarrow \texttt{Sales(i,st,s,t), i} \xrightarrow{t} \texttt{itemId:p, st} \xrightarrow{Now} \texttt{region:r.}$

The second rollup atom filters out regions not valid today.

Suppose now the following constraint: if a region where a sale occurred does not exist today, we want in the result a descendant of such region, if it exists. This query cannot be expressed in *TOLAP*. To show this, suppose a region r is split into r_1 and r_2. After that, r_1 is merged with another region r_4. In the meantime, maybe some region could have been deleted. With the tools defined so far, we could not find the "descendants" of r, because this is a transitive closure problem even though the schema remains fixed.

6.5.2 Extending *TOLAP*

We extend *TOLAP* in order to be able to express the class of queries exemplified above. First, remember that at the end of Section 5.4.5 we added two predicates to the data model, denoted $split(x, y, L, t)$ and $merged(x, y, L, t)$ in order to keep track of the different dimension's states. We defined them as *event* predicates, in the temporal database sense.

Using the *split* and *merged* predicates, we add to the syntax of *TOLAP* defined in Section 6.4.1 a new kind of atom, $\texttt{d} : \texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_1} \texttt{L}_2(\texttt{t}_2) : \texttt{y}$. The valuation of this atom proceeds as in Section 6.4.2. The interpretation is as follows: the atom evaluates to $True$ whenever y is the element in level L_2 in dimension d, to which an element x in level L_1 rolled up at time t_2, given that y is a successor(if $t_2 > t_1$) or predecessor (if $t_1 > t_2$) of an element z in L_2, s.t. x rolled up to z at time t_1.

In order to clarify the meaning of the expression $\texttt{d} : \texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_1} \texttt{L}_2(\texttt{t}_2) : \texttt{y}$ let us explain it in terms of datalog with stratified negation expressions. Let us define a predicate $shift(x, y, L, t)$ as follows:

$\texttt{shift(x,y,L,t)} \longleftarrow \texttt{split(x,y,L,t).}$

$\texttt{shift(x,y,L,t)} \longleftarrow \texttt{merged(x,y,L,t).}$

$\texttt{shift(x,y,L,t}_2\texttt{)} \longleftarrow \texttt{shift(x,z,L,t}_1\texttt{),shift(z,y,L,t}_2\texttt{)}, t_2 > t_1.$

From predicate *shift* we derive another one, called $shiftPers(x, y, L, t)$, which extends the validity of the *split* or *merge*, to every instant t between updates. For instance, if $shift(r, r_1, L, 10)$ and $shift(r_1, r_2, L, 13)$ hold, then, $shiftPers(r, r_1, L, 11)$ and $shiftPers(r, r_1, L, 12)$ also hold. This has been called *Persistence* in former temporal database works [BWJ98]. Thus:

$$
\begin{aligned}
\texttt{shiftPers(x,y,L,t)} &\longleftarrow \texttt{shift(x,y,L,t)}.\\
\texttt{shiftPers(x,y,L,s(t))} &\longleftarrow \texttt{shiftPers(x,y,L,t)},\\
&\quad \neg\texttt{shift(y,y}_1\texttt{,L,s(t))},\\
&\quad \texttt{y} \neq \texttt{y}_1, \texttt{s(t)} \leq \mathit{Now},\\
&\quad \neg\texttt{deleted(y,L,s(t))}.\\
\texttt{shiftPers(x,y,L,s(t))} &\longleftarrow \texttt{shiftPers(x,y,L,t}_1\texttt{)},\\
&\quad \texttt{deleted(y,L,t}_1\texttt{)},\\
&\quad \neg\texttt{inserted(y,L,t}_2\texttt{)},\\
&\quad \texttt{inserted(Y,L,t)},\\
&\quad \texttt{t}_1 < \texttt{t}_2, \texttt{t}_2 < \texttt{t}.
\end{aligned}
$$

Here, $s(t)$ stands for the successor of t. Predicates $deleted(y, L, t)$ and $inserted(y, L, t)$ represent the deletion of an element y from a level L containing it, or the insertion of an element into a level, respectively, and can be derived from the available data.

Now, we can define the meaning of $L_1 : x \xrightarrow{t_1} L_2(t_2) : y$ by means of the following datalog rules:

$$
\begin{aligned}
\texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_1} \texttt{L}_2\texttt{(t}_2\texttt{)} : \texttt{y} &\longleftarrow \texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_1} \texttt{L}_2 : \texttt{y},\\
&\quad \texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_2} \texttt{L}_2 : \texttt{y}.\\
\texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_1} \texttt{L}_2\texttt{(t}_2\texttt{)} : \texttt{y} &\longleftarrow \texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_1} \texttt{L}_2 : \texttt{z},\\
&\quad \texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_2} \texttt{L}_2 : \texttt{y},\\
&\quad \texttt{shiftPers(z,y,L,t}_2\texttt{)}.\\
\texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_1} \texttt{L}_2\texttt{(t}_2\texttt{)} : \texttt{y} &\longleftarrow \texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_1} \texttt{L}_2 : \texttt{z},\\
&\quad \texttt{L}_1 : \texttt{x} \xrightarrow{\texttt{t}_2} \texttt{L}_2 : \texttt{y},\\
&\quad \texttt{shiftPers(y,z,L,t}_2\texttt{)}.
\end{aligned}
$$

This implies that if at time t_1 x rolled up to z, and at time t_2 x rolled up to y, and y is a descendant of z, then use y as a rollup for x.

We will denote this extension of $TOLAP$ as $TOLAP^+$. The meaning of a $TOLAP^+$ query is analogous to the meaning of a $TOLAP$ one. Now, we can express the query *"total sales per item and region, using only the regions at which the sales occurred, or their descendants"*, as:

$$
\begin{aligned}
\texttt{IR(p,r,SUM(s))} &\longleftarrow \texttt{Sales(i,st,s,t), i} \xrightarrow{t} \texttt{itemId:p},\\
&\quad \texttt{st} \xrightarrow{t} \texttt{reg(}\mathit{Now}\texttt{):r}.
\end{aligned}
$$

In order to make things more clear, suppose that a sale such as $(p_1, s_1, 30, 10)$ occurred, and also suppose that at instant "1" store s_1 belonged to region r_2 which no longer exists at time "10",

because it was split into r_{21} and r_{22}. After a series of updates(maybe including the deletion of r_{21} and r_{22}), store s_1 currently belongs to region r_5, which is not a descendant of r_1. According to the semantics defined above, this sale will not contribute to the query result.

6.6 Summary

In this chapter we introduced the temporal query laguage for OLAP, which we called *TOLAP*. We defined the syntax and semantics of the language, and presented many examples of *TOLAP* rules and programs. Finally, we suggested a possible extension that expands *TOLAP*'s expressive power.

Chapter 7

TOLAP Implementation

In this chapter we present our *TOLAP* implementation. We describe the data structure supporting the system, and how each *TOLAP* atom is translated to SQL statements. We also show how *TOLAP* rules and programs are translated to SQL, and discuss different implementation alternatives. Finally, we apply our implementation to the case study introduced in Chapter 3.

The chapter is organized as follows. In Section 7.1 we discuss different relational representation alternatives for the temporal model. Section 7.2 presents the translation to SQL of *TOLAP* rules and programs. Section 7.3 describes the system's implementation and metadata. In Section 7.4 we apply our implementation to the medical center case study. We conclude in Section 7.5.

7.1 Relational Representation

Like in Chapter 3, two different data structures were considered for representing dimensions : a "fixed schema" versus a "non-fixed schema" approach. In both of them, a relation with schema (*dimensionId, loLevel, toLevel, From, To*) represents the structure of each dimension across time. Each tuple in this relation informs that in the dimension identified by *dimensionId*, level *loLevel* rolled up to level *toLevel* between instants *From* and *to.* We briefly discuss how instances of the dimensions are stored under both kinds of representations.

7.1.1 Fixed Schema

In a fixed-schema relational representation, a dimension instance is a relation with tuples of the form (*loLevel,upLevel,loVal,upVal,From,To*). Each tuple represents a rollup

$\rho^{upLevel}_{loLevel}[(From, To)](loVal) = upVal$. Thus, a structural dimension update will only insert new tuples in this relation, leaving the schema unaltered.

This is probably the most natural representation, because it implements the relation $\mathcal{R}_{\mathcal{D}}$ of Section 6.4.2 straightforwardly. Moreover, update operators can be easily implemented, except for *Relate* and *Unrelate*, which will require self-joining the relation, in order to check the existence of a consistency function. On the other hand, this representation has drawbacks analogous to the ones described in Chapter 3. Further, translation to SQL would be awkward, and the resulting SQL queries will be far from optimal, because the transitive closure of the rollup functions must be computed, requiring self-joining temporal relations.

7.1.2 Non-fixed Schema

In this case there is also a single relation holding the instances of each dimension, but each dimension level is mapped to an attribute in this relation. As in the denormalized representation discussed in Chapter 3, a tuple contains all the possible paths from an element in the bottom level, to the distinguished element "all". Two attributes, *From* and *To*, indicate, as usual in the temporal database field, the interval within which a tuple is valid.

This structure requires more complicated algorithms for the update operators. For instance, each generalization, specialization or level deletion, requires an update of the relation's schema. However, the translation process is simpler, and the subsequent query performance better, because the transitive closure of the rollup functions reduces to a single relation scan.

7.1.3 Fixed vs. Non-fixed Schemas

To make the ideas above more clear, let us show how a *TOLAP* query is translated to SQL. Later, we will give full details of this process.

Consider the query *"total sales by company"*, posed to the example data warehouse of Chapter 6 in which we have inserted a new level *company* above level *brand*. This query reads in *TOLAP* :

`COMP(c,SUM(qty))` $\longleftarrow$ `Sales(i,st,qty,t), i` $\xrightarrow{t}$`company:c.`

In the fixed schema representation, the dimension *Product* would be represented by relations with tuples of the form $< itemId, brand, i_1, b_1, t_1, t_3 >$, $< brand, company, b_2, c_1, t_2, t_3 >$, and so on. Thus, each time a new level is added, the relation's schema does not change. In a non-fixed

schema approach, the addition of a dimension level induces a schema update of this relation: a new column will be added to the relation. If a level is deleted, no schema update occurs(i.e., the corresponding column is not dropped), in order to preserve the dimension's history.

Assuming that no schema update affected the fact table *Sales*, the SQL equivalent of the *TOLAP* query above in the fixed schema approach will look like[1]:

```
SELECT P1.upLevel,SUM(sales)
FROM Sales S, Product P, Product P1, Time T
WHERE
  S.itemId = P.loVal AND P.loLevel = 'itemId' AND
  P.upLevel = 'brand' AND P1.upLevel = 'company' AND
  P.upLevel = P1.loLevel AND
  S.day between P.From AND P.To AND
  S.day between P1.From AND P1.To
GROUP BY P1.upLevel
```

In the non-fixed schema representation, the SQL equivalent for the query is:

```
SELECT P.company,SUM(sales)
FROM Sales S, Product P
WHERE
  S.itemId = P.itemId AND
  S.day between P.From AND P.To
GROUP BY P.company
```

It is easy to see that the result is much more elegant and efficient in the second approach. The computation of the rollup from *itemId* and *company* is straightforward, while in the first approach, the self-join of the table *Product* is required.

These arguments led us to conclude that the non-fixed schema approach would be the better choice for the current *TOLAP* implementation.

[1] For the sake of clarity we do not show here how time granularity is handled in the translation.

7.2 Translating *TOLAP* into SQL.

The *TOLAP* parser performs a first pass over a rule, checking that the syntactic conditions of Definition 25 are met. In case an error is found, execution halts. In this first pass, the different atoms are identified and a symbol table built. The second pass translates each atom into an SQL statement and builds the equivalent SQL query.

We will show how a *TOLAP* rule of the form

$$\texttt{Q(x,y,Ag(m))} \longleftarrow \texttt{F(}x_i\texttt{,}y_j\texttt{,m,t)},\ x_i \xrightarrow{t} l_i\texttt{:x},\ y_j \xrightarrow{Now} l_j\texttt{:y},\ Dim\texttt{:l:r} \xrightarrow{t} \texttt{p:z;}$$

is translated to SQL. The translation of atoms, rules, and programs will be explained in that order [2].

7.2.1 *TOLAP* Atoms

- For each *rollup atom* like $x_i \xrightarrow{t} l_i$:x, bound to a variable in a fact table, a selection clause is built as follows:

  `F.i =` $Dim_i.bottom$ `AND`

  `F.time between` $Dim_i.From$ `AND` $Dim_i.To$

  The first expression equates the bottom level of dimension Dim_i to the column of the fact table corresponding to dimension Dim_i(Dim_i is `dim(`x_i`)` in our notation above). The actual name of the column `F.i` is taken from the fact table's metadata. The second expression corresponds to the join between the fact table and the "Time" dimension.

- Each *time constant* is translated as a selection clause. The second rollup atom in the rule above will be translated as[3] :

  `F.j =` $Dim_j.bottom$ `AND`

  $Dim_j.To$ `=` Now

  Dim_j is the dimension such that variable y_j is bound to.

- If the rollup atom $x \longrightarrow$ Y:x corresponds to a user-defined time dimension, the atom is translated as

[2] **Notation:** The dimensions' sub-indices represent their position in the fact table to which they are bound. Also, constructions of the form $x \xrightarrow{t}$ Y:x, are replaced by $x[t] \longrightarrow$ Y:x in the implementation, in order to simplify parsing.

[3] In an actual implementation, *Now* can be replaced by *Sysdate*() or any function returning the current time.

F.j = $Dim_j.bottom$

The user-defined time dimension is identified by the *dtime* attribute in the catalog table DIMENSION.

- A rollup atom Dim:l:r $\xrightarrow{t}$ p:z, not bound to any fact table, is translated as an EXISTS clause.

```
EXISTS
  SELECT *
  FROM Dim
  WHERE
   F.Time between Dim.From AND Dim.To
```

The WHERE clause is omitted if the join with the time dimension is not required.

- The rollup from the bottom levels of the dimensions to the levels l_i and l_j corresponding to the variables in the head of the rule above (the aggregation levels), is computed as a projection and an aggregation in the way shown below. Thus, the SQL query generated by the *TOLAP* query of the beginning of this section, will look like

```
SELECT Dim_i.l_i, Dim_j.l_j,Ag(measure)
FROM F_1, Dim_i, Dim_j
WHERE
  F_1.i = Dim_i.bottom AND
  F_1.j = Dim_j.bottom AND
  F_1.time between Dim_i.From AND Dim_i.To AND
  Dim_j.To = Now AND
  EXISTS
    (SELECT *
    FROM Dim
    WHERE
    F_1.Time between Dim.From AND Dim.To )
GROUP BY Dim_i.l_i, Dim_j.l_j
```

The term measure is the measure in the fact table, bound to variable m. The fact table sub-index represents the version of the fact table. In this case there is only one version, as

no schema update occurred. We will come back to this shortly.

- A *constraint atom* is translated as a selection condition in the `WHERE` clause. If the constraint atom is *negated,* this condition is treated in the usual way (a `NOT` predicate is added).

- A *negated rollup atom* is translated as a `NOT EXISTS` clause. Suppose that in the query above we add a negated atom as follows [4]:

  `!(`$y_j \stackrel{Now}{\longrightarrow} l_1$`:'a')`

  where 'a' represents a constant. This atom is converted into an SQL expression of the form:

  ```
  NOT EXISTS(
    SELECT *
    FROM Dim_j
    WHERE
    Dim_j.To = Now AND     Dim_j.l_1 ='a' AND F.j = Dim_j.bottom)
  ```

  where l_1 is the attribute representing level l_1.

- A *predicate atom* is translated as a table in the `FROM` clause, with the conditions which arise from the variables or constants in it.

7.2.2 *TOLAP* Rules

So far we tackled the problem of translating each atom in a *TOLAP* rule separately. The next step will be the study of the translation of the whole rule.

Above, we gave an example of a simple rule, assuming no schema update occurred in the fact table. However, we claimed that one of the main features of *TOLAP* is the ability to deal with schema updates triggered by a specialization or the deletion of a bottom level. In the table META_FACT_TABLE (see Subsection 7.3.3), a tuple is stored each time a dimension update affects a fact table in the data warehouse. Given a fact table F, this fact table may have versions F_1, F_2 and so on, each one with different schemas.

Given a *TOLAP* query Γ involving a fact table F, with versions $F_1, \ldots, F_n$, the SQL query Q equivalent to Γ will be such that

[4] Actually, the parenthesis is not required

$$Q \quad = \quad Q_1 \cup Q_2 \cup \ldots \cup Q_n,$$

where Q_i are queries involving facts occurred in the intervals I_i in which each F_i holds. If the query involves an aggregation function, call it f_{AGG}, one more aggregation must be performed, in order to consider duplicates in each subquery. Thus

$$Q_{AGG} \quad = \quad f_{AGG}(Q)$$

7.2.3 *TOLAP* Programs

TOLAP programs are treated as a series of *TOLAP* rules. Let us consider again the *TOLAP* program of Subsection 6.3.3 above: "total sales by brand, for those brands that sold more than one hundred thousand dollars in Buenos Aires".

```
BASales(c,SUM(m)) ⟵  Sales(a,s,m,t), a -t->brand:c,
                     s -t->city:x, x.name=''Buenos Aires;''

Q(c,SUM(m)) ⟵  Sales(a,s,m,t), BASales(c,q), a -t->brand:c, q ≥ 100000;
```

The program is processed as follows: during parsing, a temporary table is created for each predicate in the head of the first rule (the semicolon indicates the end of each rule). This predicate is entered in the symbol table, so it can be accepted when parsing the second rule. Then, the program generates two SQL queries. The first one populates the temporary table, and the second one computes the final output of the program. All temporary tables are dropped after program execution.

7.2.4 Query Optimization

There are cases in which the join between dimensions and fact tables is not needed. This situation arises when a variable in the head of the rule belongs to a level which is the bottom level of a fact table in the body (or was the bottom level at least during some interval I_i). Our implementation takes advantage of this fact, and does not generate the join.

Example 40 *Let us consider again the situation discussed in Section 6.3.2. The fact table* Sales *will be split into Sales_1 and Sales_2, each one holding before and after d_5. Let us analyze how the*

query "total sales by itemId and city" will be translated to SQL. The query in TOLAP reads:

```
IC(it,ci,SUM(m))  ⟵  Sales(i,s,m,t), i -t-> itemId:it,
                     s -t-> city:ci.
```

Two SQL subqueries will be generated, one for each fact table.

```
SELECT itemId,city,SUM(sales)
FROM (
  SELECT itemId,city, SUM(sales)
  FROM Sales_1
  GROUP BY itemId,city

  UNION ALL

  SELECT itemId,city, SUM(sales)
  FROM Sales_2,Store
  WHERE
  Sales_2.Time between Store.From AND Store.To AND
  Sales_2.storeId=Store.storeId
GROUP BY itemId,city )
GROUP BY itemId,city
```

Notice that, as city *and* itemId *were the bottom levels of* `Sales_1` *no join is needed in the first subquery.*

If a *TOLAP* rule contains a constraint atom with a condition over time such that the lifespan of a version of the fact table does not intersect with the interval determined by the constraint, the subquery corresponding to that version of the fact table is not generated by the translator, as it will not yield any tuple in the output. For instance, in Example 40, adding the constraint $t < d_6$ will prevent the first subquery from being generated. We call this step *subquery pruning.*

7.3 Implementation

In this section we briefly describe our implementation of *TOLAP*. We focus on describing the system's metadata, because it constitutes the heart of the implementation.

7.3.1 Implementation Tools

The system was implemented on an ORACLE 8.04 database. The parser and visual interfaces were written in Java using Borland's Java Builder. The update operators were implemented as ORACLE's PL-SQL stored procedures and functions.

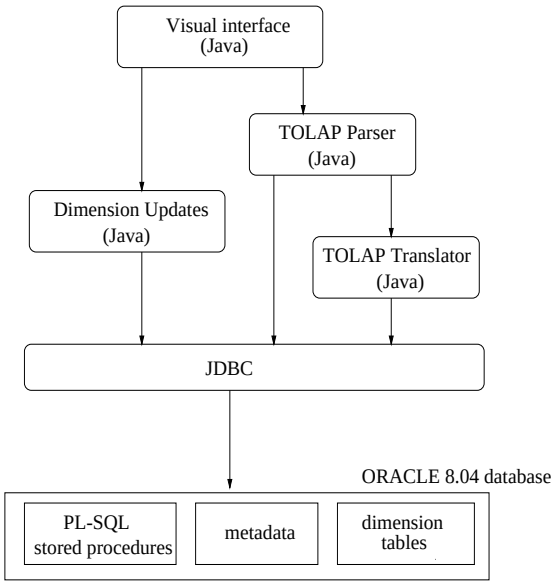

Figure 7.1: System's architecture

7.3.2 Architecture

Figure 7.1 shows the system's architecture. The user interacts with the system through the visual interface. The dimension updates and the *TOLAP* compiler access the database through a JDBC driver. The compiler uses stored procedures when processing metaqueries.

7.3.3 Metadata

Besides the relations representing each dimension in the data warehouse, several tables are needed in order to store the system's metadata. We briefly describe the main ones.

- DIMENSION(dimensionId,description,granularity,init,dtime). Holds the basic information of each dimension in the system. Attribute *init* indicates if the user is working in "temporal" or "non-temporal" mode. This is useful when the user does not want to record a dimension's history until a stable stage has been reached. For instance, the user would like to start from a dimension *Product* with levels *itemId, brand* and *category*. In this case, the user indicates that she wants to work in the "initial mode", and proceeds to create the dimension. When the dimension is completed, the user shifts back to the "temporal mode". The value of attribute *init* ('I' or 'T' for 'initial' and 'temporal' respectively) is reset each time the user shifts modes. The default is 'T'.

Attribute *dtime* informs if the dimension is a user-defined time dimension. This information becomes necessary when translating *TOLAP* rules to SQL (see Subsection 7.2).

- DIMENSION_H(dimensionId,TmpFrom,TmpTo,LevelFrom,LevelTo). Represents the history of the dimension schema.
- META_FACT_TABLE(FactTableId,From,To,dimension,number,granularity, ...). Keeps information of the different stages of the fact tables in the system, and the correspondence of dimensions to columns in the fact table.
- ATTRIBUTE_LEVELS(dimensionId,level,attributeName). Holds the names of the attributes of each level.
- ATTRIBUTE_INSTANCE(dimensionId,level,value,attrib,valueAt). Holds the names and values of the attributes of each level. The column *value* is the element in level *level* which is described by attribute *attrib* with value *valueAt*.

  Note. In the current version of *TOLAP*, attributes are not temporal objects. Thus using the notation of table ATTRIBUTE_INSTANCE, *valueAt* is constant for all the lifespan of *value*.

7.4 The Medical Clinic Case Study: a Temporal Approach

In this section we apply the temporal approach to the Case Study introduced in Chapter 3. The temporal multidimensional data model allows not only to modify the dimensions in an on-line fashion, but to keep track of the history of the medical data warehouse. We will present a simulated scenario, based on real data and the available information already described in Section 4.

7.4.1 Goals of the Study

We will study, on a real-life case, if the temporal approach and the *TOLAP* query language address the needs of the users in a better way than commercially available non-temporal OLAP tools. Using our implementation, we will discuss: (a) performance, measuring the response time of the dimension updates and *TOLAP* queries;(b) expressiveness and abstraction, posing queries which could not be easily expressed in non-temporal environments and comparing these queries with their SQL equivalent;(c) visualization capabilities, through a graphic environment which allows browsing dimensions and fact tables across time, applying dimension updates, and querying the temporal data warehouse.

Table	# of tuples
Patient	16383
Procedure	11253
Doctor	822
Time	730
Services_1	26040
Services_2	12624
Services_3	17828

Figure 7.2: Number of tuples in the temporal data warehouse tables

7.4.2 Data Preparation

The testing scenario was prepared as follows:

1. We created the temporal data warehouse dimensions using a sequence of temporal dimension updates. We also created the initial version of the fact table; the other two versions were triggered by the dimension updates. The dimension updates were applied using the graphic environment we describe below.

2. After this, we populated (off-line) the three versions of the fact table, with a subset of the data used to populate the fact table *Services* in Chapter 4. The number of tuples in each table of the relational representation after all these operations were performed, is showed in Figure 7.2.

The temporal data warehouse was created as follows:

- *Procedure.* Created with bottom level *procId.* Subsequent operations performed using the graphic interface generalized this level to levels *subgroup* and *procType.* Finally, *subgroup* was generalized to *group,* and *procType* related to *group.* These dimension levels were discussed in Chapter 3.

- *Patient.* Here, the initial bottom level *patientId* represents information about the person under treatment. This level was generalized to *yearOfBirth, gender* and *institution,* in that order. Levels *yearOfBirth* and *institution* were further generalized into *yearRange* and *instype* respectively.

- *Doctor.* In order to make the study more interesting, we assumed that although facts were recorded at the *idDoctor* level, this level was temporarily deleted, triggering fact table evolution. Thus, during a short interval, the dimension's bottom level was *speciality,* later specialized into level *doctorId.*

- A user-defined *Time* dimension, with granularity *day*, allowing to express aggregates over time. The dimension's hierarchy is the following: $\{day \preceq week,\ day \preceq month,\ week \preceq All,\ month \preceq All\}$. In the relational model, the dimension is a relation with schema $\{day, week, month, year\}$, and a tuple in an instance of this relation has the form $< 10, 2, 1, 1 >$, meaning that day '10' belongs to week '2', month '1' and year '1'. Attributes could be added, for instance, to cover the case of a user wishing to find out the number of services delivered on holidays. For the present study we did not define attributes for the *Time* dimension. Also, the fact table was populated in a way such that the default time dimension(with granularity *second*) and the user defined dimension (i.e. *Time*, with granularity *day* as we explained above) represent *valid* time. Notice, however, that other applications could require that bi-temporal management. In such cases, one of the time dimensions could be defined as representing *transaction* time, with the other one representing *valid* time.

In summary, the sequence of data warehouse updates was:

1. Create dimension *Doctor* with bottom level *idDoctor.*
2. Create dimension *Patient* with bottom level *patientId.*
3. Create dimension *Procedure* with bottom level *procId.*
4. Create dimension *Time* with bottom level *day.*
5. Generalize *idDoctor* to *speciality.*
6. Generalize *procId* to *subgroup.*
7. Generalize *procId* to *procType.*
8. Generalize *patientId* to *yearOfBirth.*
9. Generalize *patientId* to *gender.*

10. Generalize *subgroup* to *group.*

11. Generalize *patientId* to *institution.*

12. The Fact Table *Services*(named *Services_1*) was created, with bottom levels *procId, patientId, idDoctor* and *day.*

13. Generalize *institution* to *instType.*

14. Generalize *yearOfBirth* to *yearRange.*

15. Relate levels *practiceType* and *subgroup.*

16. Delete Level *idDoctor.*

    The second version of *Services* (denoted *Services_2*) was created, with bottom levels *procId, patientId, speciality* and *day.* This action is triggered by the *Delete Level* operation.

17. Specialize *speciality* into *doctorId.*

    The third version of *Services* was created, with bottom levels *procId, patientId, doctorId* and *day.* As above, the action was triggered by the dimension update.

In the meantime, we performed several insertions, deletions, splits, merges and reclassifications simulating situations where new doctors were hired, others left, new procedures or specialities were created, or patients moved from one institution to another. We have already shown that these situations are adequately captured by the model and *TOLAP.*

7.4.3 Queries

Figure 7.3 shows one of the sets of *TOLAP* queries we ran. The objective was finding out how the number of dimensions involved in the query affects query performance. Query Q1 has three rollup atoms in the body, while query Q3 has only one of such atoms. Moreover, we also ran the three queries replacing variable t by the constant Now (i.e., the current values of the rollups are considered for aggregation, instead of the values holding at the time of the service). For instance, query Q3 was modified to :

```
Q(a,b,c,SUM(m))  ⟵  Services(do,pr,pat,d,m,t), pa[Now] → instype:b;
```

```
Q1:  Q(a,b,c,SUM(m))  ⟵  Services(do,pr,pat,d,m,t), do[t] → speciality:a,
                         pa[t] → instype:b, pr[t] →subgroup:c;
Q2:    Q(b,c,SUM(m))  ⟵  Services(do,pr,pat,d,m,t), pr[t] →subgroup:c,
                         pa[t] → instype:b;
Q3:      Q(b,SUM(m))  ⟵  Services(do,pr,pat,d,m,t), pa[t] → instype:b;
```

Figure 7.3: Queries

Finally, we included a constraint atom in the three queries, to see the influence of the subquery pruning step. The constraint $t <$ ``02/08/2001'' leaves out fact tables *Services_2* and *Services_3*, while the constraint $t <$ ``02/13/2001'' leaves out fact table *Services_3*. For instance, query Q3 was modified as follows:

```
Q(a,b,c,SUM(m))  ⟵  Services(do,pr,pat,d,m,t), pa[t] → instype:b,
                    t <``02/13/2001'' ;
```

7.4.4 Hardware

The hardware support for these experiments was the same as the one used for the tests described in Chapter 3.

7.4.5 Discussion of Results

Performance. Figure 7.4 shows the query execution times for the three sets of queries described above. Each query was ran three times, and the average response time is displayed in the table, expressed in seconds. The numbers between parentheses represent the number of tuples in the query result. We see that replacing the time variable t with the constant Now is not relevant. However, subquery pruning reduces execution times by a factor between two and four in this example. Of course, this will depend on the size of the pruned fact table. As we could have expected, query Q1 takes longer to complete than the other two, as a multiway join of four tables is performed.

The table of Figure 7.5 shows the execution times for the dimension updates, executed in the order detailed above. We can see that they are compatible with a real environment. Note that when the same level is generalized several times, response gets slower because the size of the table increases in order to keep the history of the dimension. We did not record results for the dimension

	Query type	Q1	Q2	Q3
1	t	290 (977)	130 (361)	40 (4)
2	$t = Now$	250 (977)	110 (361)	30 (4)
3	$t <$"02/08/2001"	60 (289)	50 (95)	15 (4)
4	$t <$"02/13/2001"	140 (620)	125 (230)	20 (4)

Figure 7.4: Query execution time

Time because they are not relevant, as the dimension has fixed size during the lifespan of the data warehouse.

Table in Figure 7.6 shows the results of a second test, carried out as follows: for the dimension *Patient* we performed a sequence of dimension updates, each one increasing the number of tuples in the dimension. In order to compare this solution with the non-temporal approach, we included the results obtained for an equivalent non-temporal dimension *Patient*(created in *initial* mode, which does not account for the dimension's history). In Figure 7.6, we indicate these results in the rightmost two columns.

Expressiveness. The queries below show *TOLAP*'s expressive power applied to this case study. These queries cannot be expressed in a non-temporal model without ad-hoc design. The temporal approach shows its power in queries which compare the times of occurrence of different events.

For example, let us suppose a user wants to analyze the workloads of doctors, in order to estimate future needs. The following query could be used for measuring how the the arrival of a new doctor influences the number of patients served by a doctor named *Roberts*:

"List the total number of services delivered weekly by Doctor Roberts while Doctor Richards was not working for the clinic."

```
patRob(w,SUM(m))  ⟵  Services(do,pr,pat,d,m,t), do ⟶[t] doctorId:d,
                     d.name=''Roberts'', d ⟶ week:w
                     !Doctor:doctorId:dd⟶[t]All:all,dd.name=''Richards''.
```

Notice in this query that the negated atom is not bound to the fact table. Also notice the use of the user-defined *Time* dimension. The following query returns the number of services delivered

Table	Time (sec)
Generalize *idDoctor* to *speciality*	2
Generalize *procId* to *subgroup*	10
Generalize *procId* to *procType*	320
Generalize *patientId* to *yearOfBirth*	190
Generalize *patientId* to *gender*	280
Generalize *subgroup* to *group*	10
Generalize *patientId* to *institution*	320
Generalize *institution* to *instType*	5
Generalize *yearOfBirth* to *yearRange*	5
Relate procType to group	65
Delete Level *idDoctor*	30
Specialize *speciality* to *doctorId*	2

Figure 7.5: Dimension updates execution time

by Dr. Roberts while both doctors were hired by the clinic.

```
patRob(w,SUM(m)) ⟵  Services(do,pr,pat,d,m,t), do -t-> doctorId:d,
                    d.name=''Roberts'',d --> week:w
                    Doctor:doctorId:dd-t->All:all,dd.name=''Richards''.
```

Figure 7.7 shows the translation to SQL of the first query. For the sake of clarity we only display in full the portion of the query corresponding to the first fact table state. The other parts are solved in a similar fashion. Note the complexity of the SQL query, compared with its *TOLAP* equivalent.

The next query illustrates how to check patients who were served when they were affiliated to 'MEDICUS' and are currently affiliated to 'SWISS MEDICAL'.

```
changePlan(pat) ⟵  Services(do,pr,pat,d,m,t),pat -t-> institution:'MEDICUS',
                   pat -Now-> institution:'SWISS MEDICAL'.
```

Below, we give another query that cannot be expressed in non-temporal systems.

"Was the clinic giving cancer treatment service while patient 'John Ash' was registered?."

# tuples in dimension	Update	Time (sec)	# tuples (nt)	Time (sec)(nt)
2727	create dimension	2	2727	2
2727	Gen. *patientId* to *gender*	20	2727	20
5452	Gen. *patientId* to *age*	30	2727	18
8179	Gen. *patientId* to *institution*	230	2727	20
10906	Gen. *age* to *ageRange*	30	2727	10
13633	Delete *gender*	15	2727	3
16356	Delete *institution*	20	2727	3
21808	Delete *age*	25	2727	3

Figure 7.6: Dimension updates for dimension *Patient*, temporal and non-temporal

```
cancTr() ⟵  Procedure:group:x ⟶ᵗ All:all,
            x.desc=''cancer treatment'', Patient:patientId:y ⟶ᵗ All:all,
            y.name='John Ash'.
```

7.4.6 Visualization: a Walk-through

In this subsection we will conduct a tour through the graphic environment developed taking advantage of the temporal multidimensional model.

The graphic interface allows to:

- browse dimensions across time, and watch how they were hierarchically organized throughout their lifespan. Further, dimension instances could also be browsed.
- apply all the basic and complex dimension updates;
- import rollup functions from text files;
- browse different versions of a fact table;
- send *TOLAP* programs to the system, and display their results without leaving the environment, including seeing the generated SQL query.

Figure 7.8 shows a typical system screen depicting dimension *Patient* as of December 13th, 2000. The window on the left presents the dimension's structure. The arrows upon the window

```
 patRob(w,SUM(m))  ⟵  Services(pr,pat,do,d,m,t), do[t] ⟶ doctorId:d,
                      d.name=‘‘Roberts’’,d ⟶ week:w
                      !Doctor:doctorId:x[t]⟶ All:‘‘all’’,x.name=‘‘Richards’’.
```

```
SELECT COL_0,SUM(SUM_0)
  FROM (
    SELECT DIMENSION_TIME.WEEK COL_0, SUM(AMNT) SUM_0
    FROM SERVICES_1 SERVICES_1, DIMENSION_DOCTOR, DIMENSION_TIME,
    WHERE NOT EXISTS (
     SELECT * FROM SERVICES_1 FT, DIMENSION_DOCTOR
     WHERE to_date(to_char(FT.Tmp,'dd/mm/yyyy hh24'),'dd/mm/yyyy hh24')BETWEEN
     to_date(to_char(DIMENSION_DOCTOR.tmp_from,'dd/mm/yyyy hh24' ),'dd/mm/yyyy hh24' )
     AND decode(to_date(to_char(DIMENSION_DOCTOR.tmp_to,'dd/mm/yyyy hh24'),
     'dd/mm/yyyy hh24'),null, to_date(to_char(sysdate,'dd/mm/yyyy hh24'),
     'dd/mm/yyyy hh24' ),to_date(to_char(DIMENSION_DOCTOR.tmp_to,'dd/mm/yyyy hh24' ),
     'dd/mm/yyyy hh24')) AND to_date(to_char(FT.Tmp,'dd/mm/yyyy hh24'),
     'dd/mm/yyyy hh24')= to_date(to_char(SERVICES_1.Tmp,'dd/mm/yyyy hh24'),
     'dd/mm/yyyy hh24' )
     AND DIMENSION_DOCTOR.IDDOCTOR IS NOT NULL
     AND DIMENSION_DOCTOR.ALL = 'all'
     AND UPPER(DIMENSION_DOCTOR.IDDOCTOR) IN
       (SELECT UPPER(value) FROM ATTRIBUTE_INSTANCE
       WHERE LTRIM(RTRIM(UPPER(dimension_id))) = LTRIM(RTRIM('D_DOC_88'))
       AND LTRIM(RTRIM(UPPER(level))) = LTRIM(RTRIM('IDDOCTOR'))
       AND LTRIM(RTRIM(UPPER(attrib))) = LTRIM(RTRIM('NAME'))
       AND valueAt='Richards' ))
       AND SERVICES_1.DAY=DIMENSION_TIME.DAY
       AND DIMENSION_TIME.WEEK IS NOT NULL
    AND to_date(to_char(SERVICES_1.tmp,'dd/mm/yyyy hh24' ),'dd/mm/yyyy hh24' ) BETWEEN
    to_date(to_char(DIMENSION_DOCTOR.tmp_from,'dd/mm/yyyy hh24' ),'dd/mm/yyyy hh24')
    AND decode( to_date(to_char(DIMENSION_DOCTOR.tmp_to,'dd/mm/yyyy hh24'),
    'dd/mm/yyyy hh24'), null, to_date(to_char(sysdate,'dd/mm/yyyy hh24'),
    'dd/mm/yyyy hh24' ), to_date( to_char(DIMENSION_DOCTOR.tmp_to,'dd/mm/yyyy hh24' ),
    'dd/mm/yyyy hh24' ))
    AND UPPER(DIMENSION_DOCTOR.IDDOCTOR) IN
      (SELECT UPPER(value) FROM ATTRIBUTE_INSTANCE
      WHERE LTRIM(RTRIM(UPPER(dimension_id))) = LTRIM(RTRIM('D_DOC_88'))
      AND LTRIM(RTRIM(UPPER(level))) = LTRIM(RTRIM('IDDOCTOR'))
      AND LTRIM(RTRIM(UPPER(attrib))) = LTRIM(RTRIM('NAME'))
      AND valueAt='Roberts' )
    AND SERVICES_1.DOCTORID = DIMENSION_DOCTOR.DOCTORID
    GROUP BY DIMENSION_TIME.WEEK
  UNION ALL
    SELECT DIMENSION_TIME.WEEK COL_0, SUM(AMNT) SUM_0
    FROM SERVICES_2 SERVICES_2, DIMENSION_DOCTOR, DIMENSION_TIME
    WHERE NOT EXISTS (SELECT * FROM SERVICES_2 FT, DIMENSION_DOCTOR
    ...
  UNION ALL
    SELECT DIMENSION_TIME.WEEK COL_0, SUM(AMNT) SUM_0
    FROM SERVICES_3 SERVICES_3, DIMENSION_DOCTOR, DIMENSION_TIME
    WHERE NOT EXISTS (SELECT * FROM SERVICES_3 FT, DIMENSION_DOCTOR
    ...)
GROUP BY COL_0
```

Figure 7.7: Translation of a *TOLAP* Rule with Negation

allow browsing forward and backward across time. This is not allowed if the system is in 'initial' mode. The window on the right shows the dimension's instances. This window is synchronic with respect to the one on the left. Thus, the instances being displayed correspond to the structure on the left although they can vary as instance updates occur. However, while the screen of the left remains unchanged, several instance updates may occur, which can be displayed in the window on the right. The little box in the upper middle indicates the number of elements in the bottom level which are going to be displayed, allowing partial loading of the instance graph in main memory. This feature is crucial in making the tool usable in real applications, because loading the entire instance graph would be very expensive even for not very large dimensions.

In Figures 7.8 to 7.19 we give a graphic description of the systems's functionalities using our case study. Figures 7.8 to 7.10 show a typical browsing through time of the dimension *Patient,* depicting how the schema and instances of the dimension evolve. Figures 7.11 to Figures 7.19 show how dimensions and fact tables are created and updated.

Figure 7.11 shows how how dimension *Patient* was created, with bottom level *patientId.* The creation time is shown, the default being the current time. The *Data Table* field lets the user specify the table from which data in the bottom level will be loaded. This table could be loaded from a text file using a system provided loader, invoked clicking on the *Tables* menu.

Figures 7.12 and 7.13 deal with fact tables. Figure 7.12 illustrates the fact table creation procedure for the fact table *Services.* Just choose one or more dimensions displayed on the left window of the form, and load them into the box on the right. If the granularity of one dimension is not compatible with the granularity of the fact table, the fact table is not created and an error message displayed. Figure 7.13 shows the different versions of the fact table *Services* before the last specialization occurred.

The menu in Figure 7.14 is displayed after right-clicking on the level to be updated. Choosing the "Attributes" option allows defining a new attribute for the level (Figure 7.15). Right-clicking on an *element* of a level (in the right window), and choosing the "Attributes" option, opens the window of Figure 7.16, where the user can define the value for an attribute of the level corresponding to the selected instance. In Figure 7.16, we are defining the name of Patient number 52765.

Figure 7.17 shows the window that appears when choosing the *Specialize* operation after having clicked the right button over the dimension level *speciality.* Also, clicking the right button of the mouse over the screen on the right and choosing "AddInstance", opens the screen of Figure 7.18.

Here, we are inserting Patient number 77789, male, born in 1997 affiliated to Institution 138. Analogously we proceeded with the *merge* operator depicted in Figure 7.19. After shadowing the elements to be merged(on the left window of the form), we have loaded them into the right window. Clicking the 'Save' button will perform the update.

7.5 Summary

In this chapter we have described our *TOLAP* implementation, giving details of the process of translating a *TOLAP* rule or program into SQL. This translation makes it clear how simple it is to express in *TOLAP* a query that in SQL takes many lines of complex code. Finally, we applied the implementation to the case study introduced in Chapter 3, showing that *TOLAP* can be useful for a real-life application, overriding the limitations of non-temporal OLAP commercial tools.

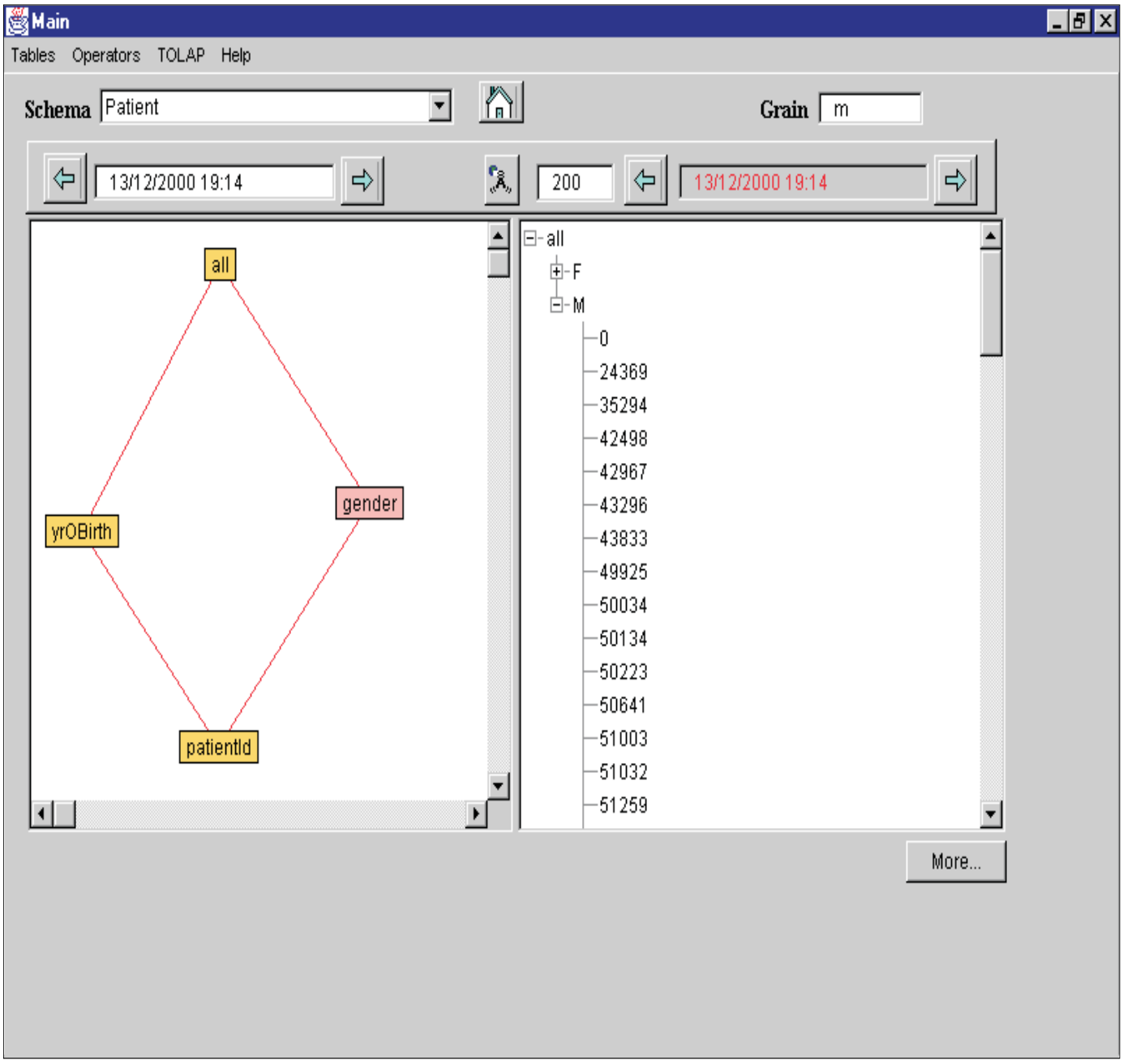

Figure 7.8: Browsing dimension *Patient* (1)

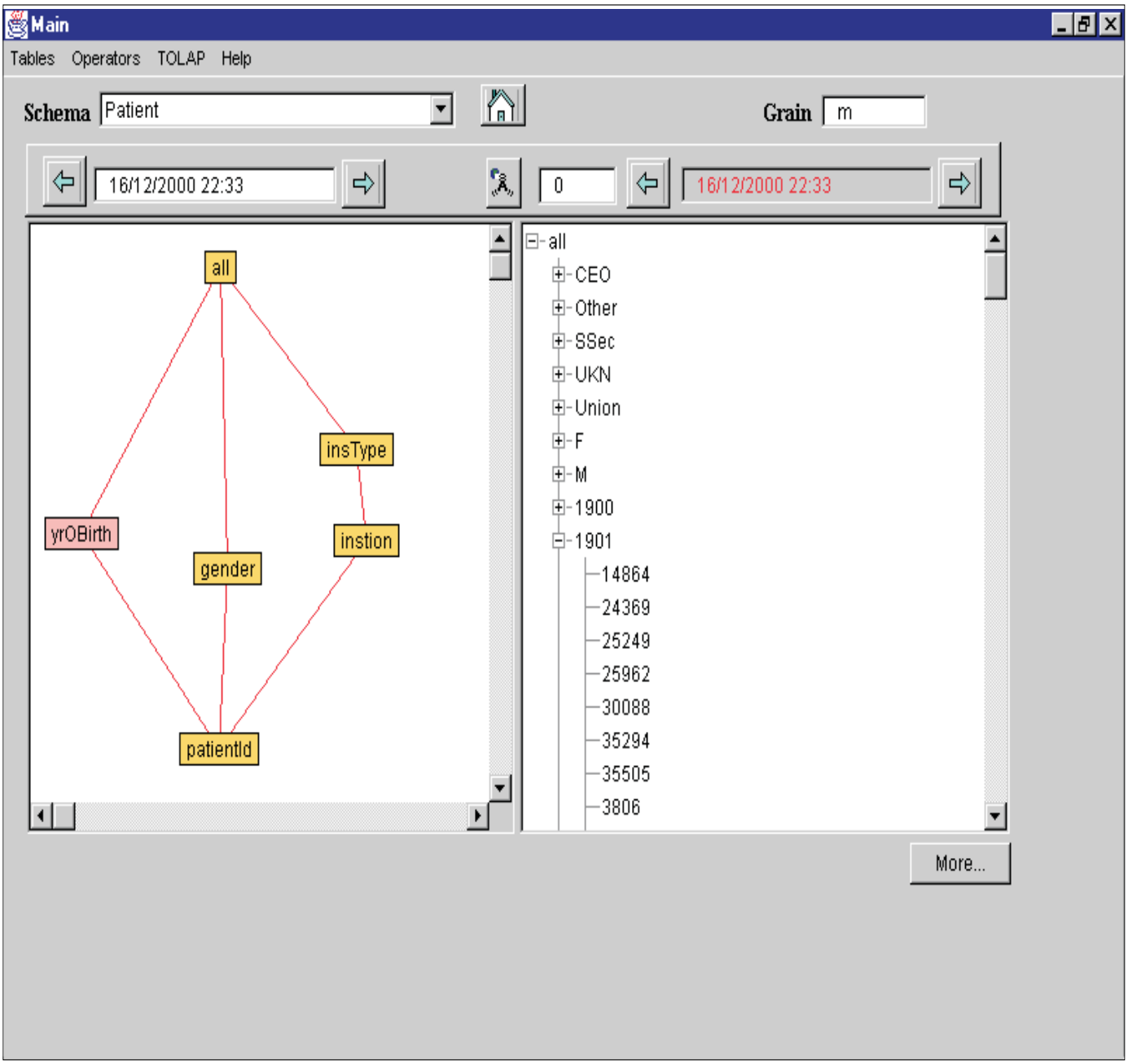

Figure 7.9: Browsing dimension *Patient* (2)

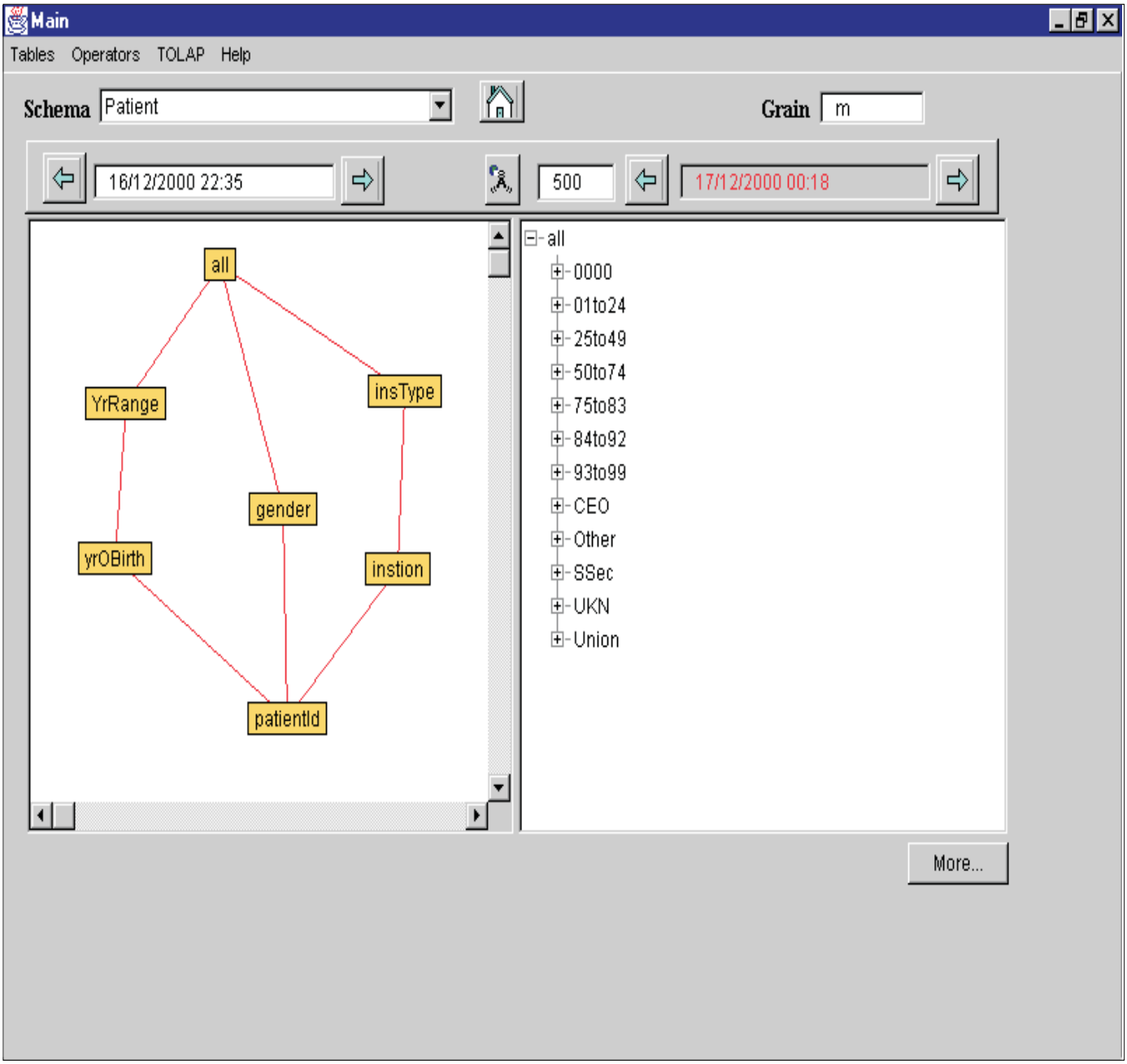

Figure 7.10: Browsing dimension *Patient* (3)

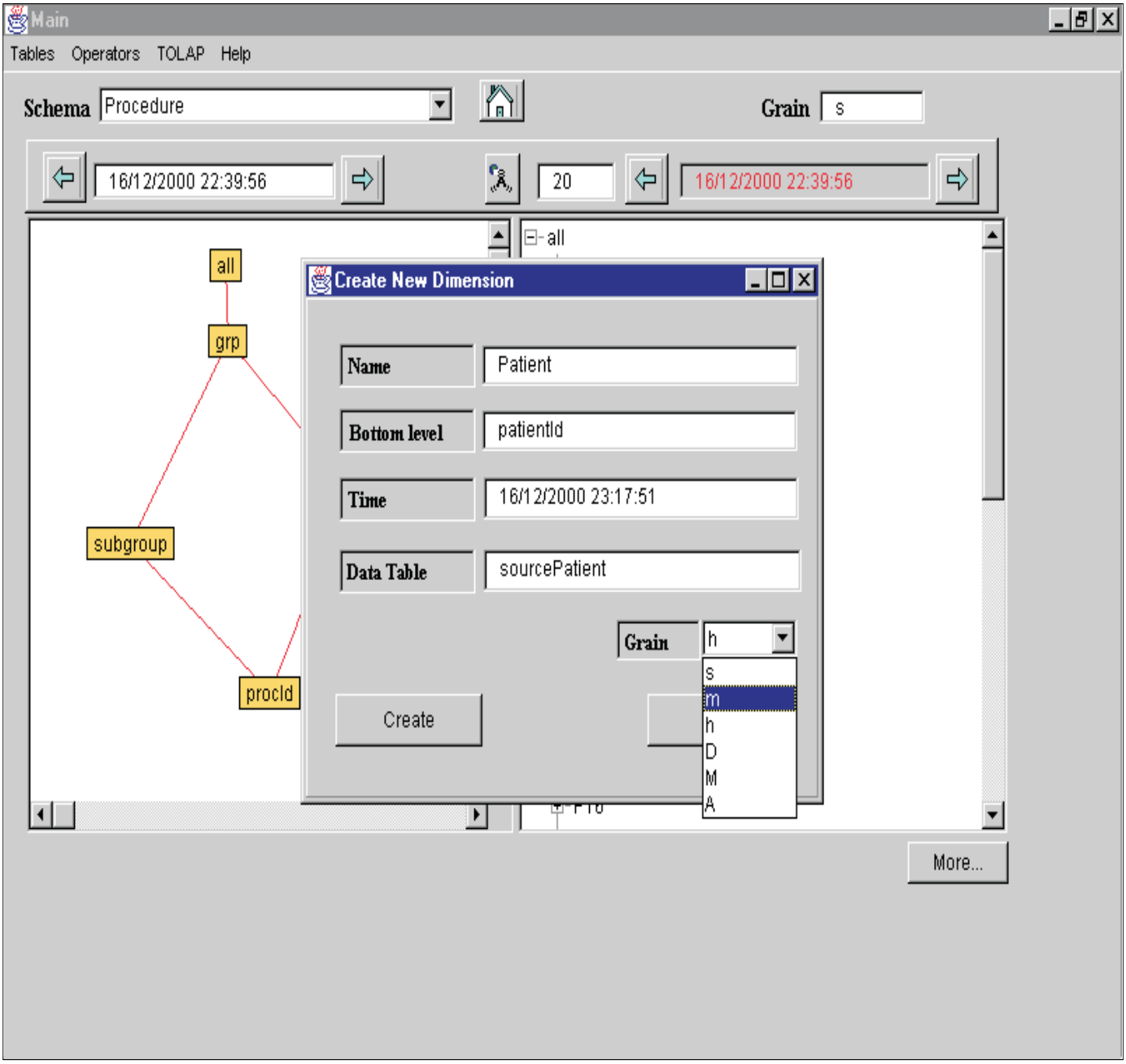

Figure 7.11: Creating a new dimension

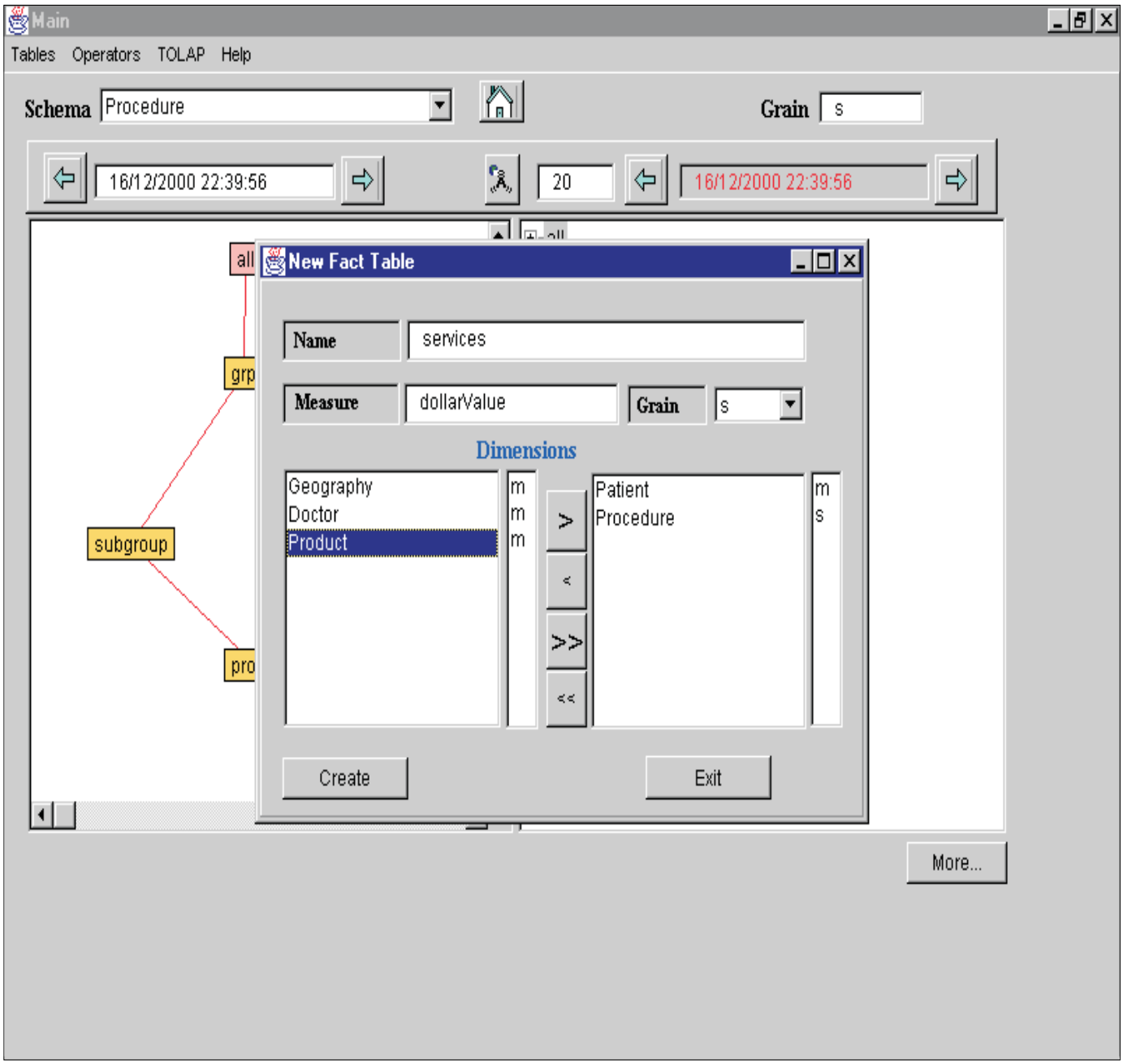

Figure 7.12: Creating a new fact table

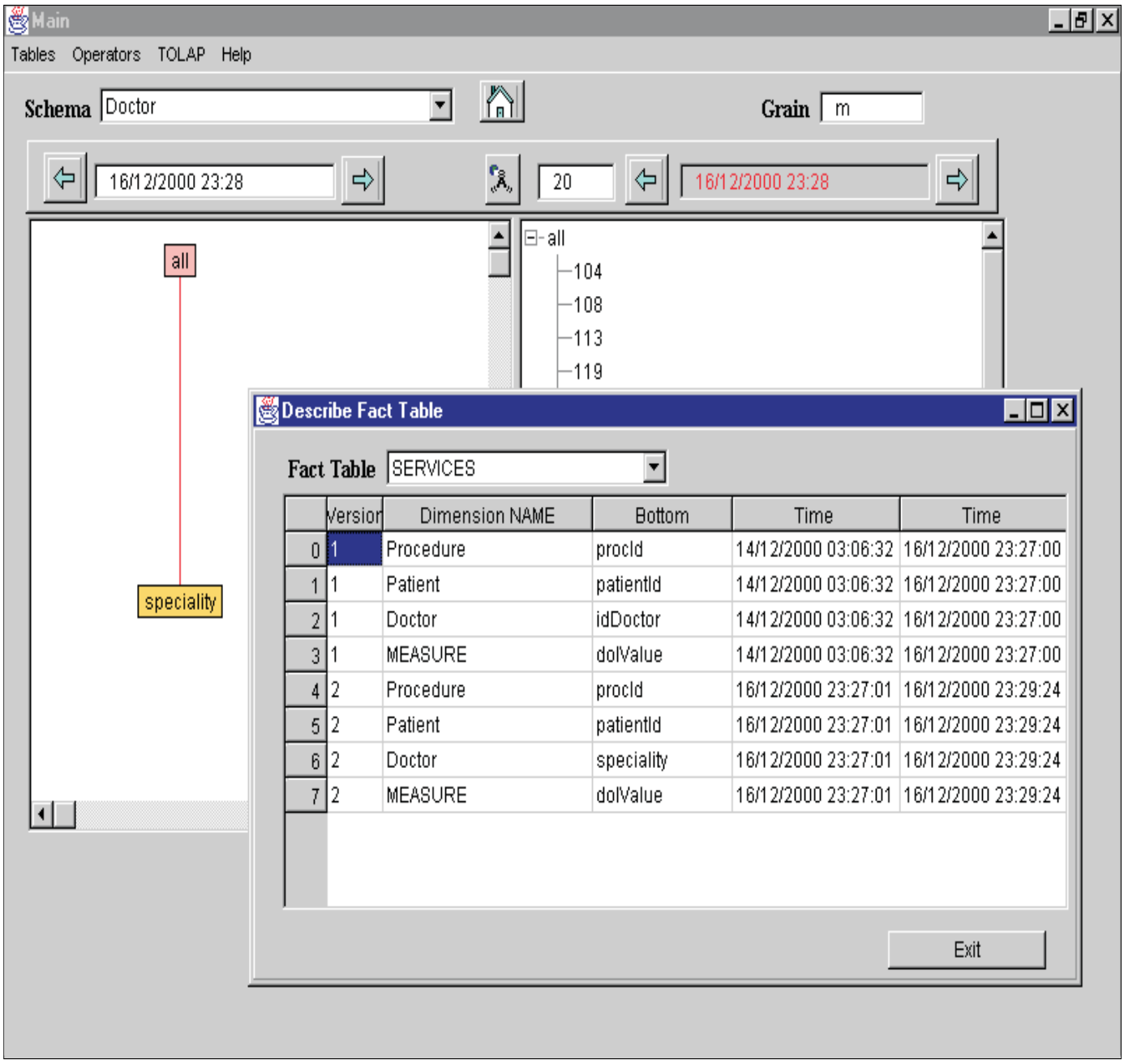

Figure 7.13: Describing a fact table

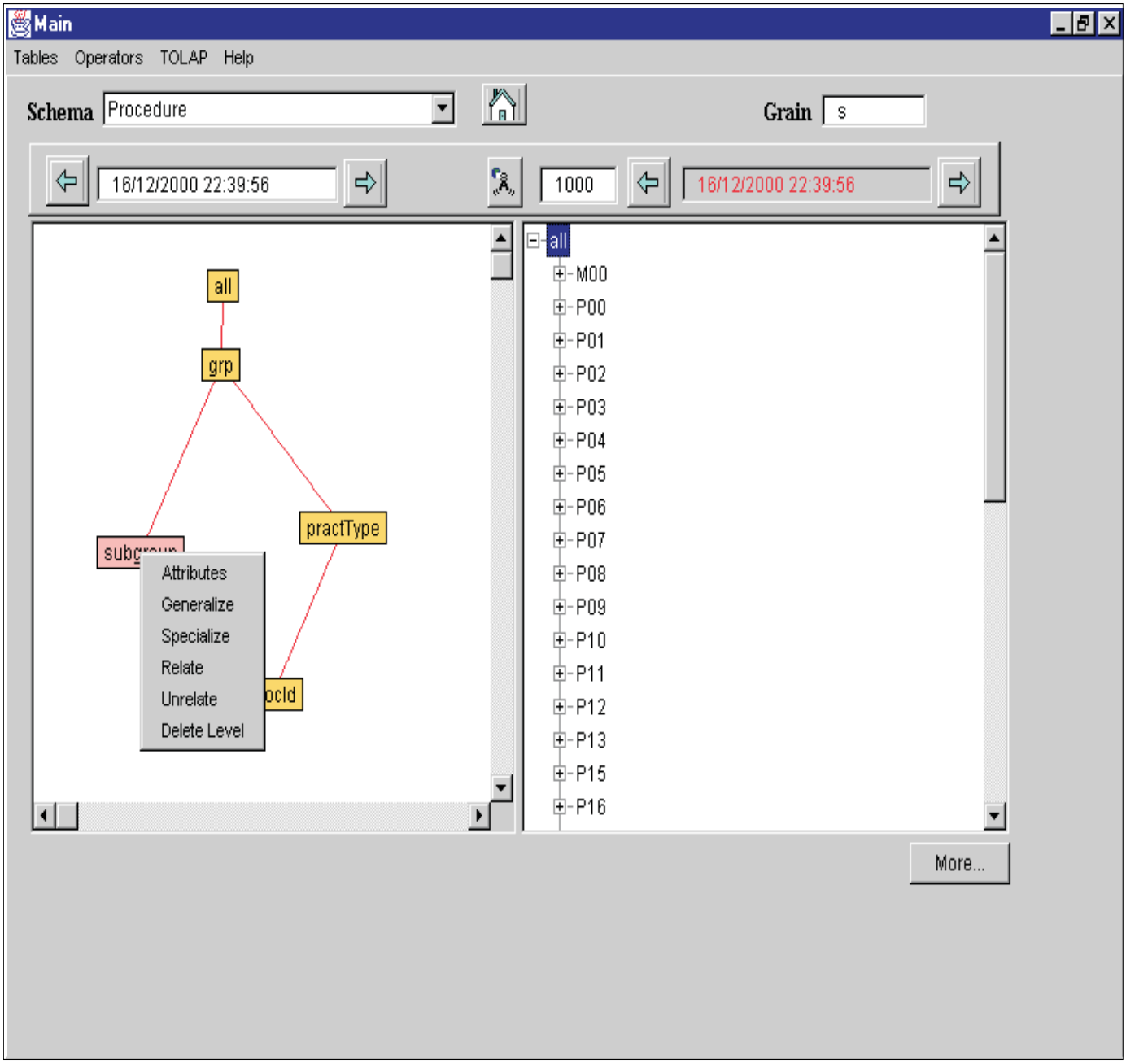

Figure 7.14: Applying structural operators

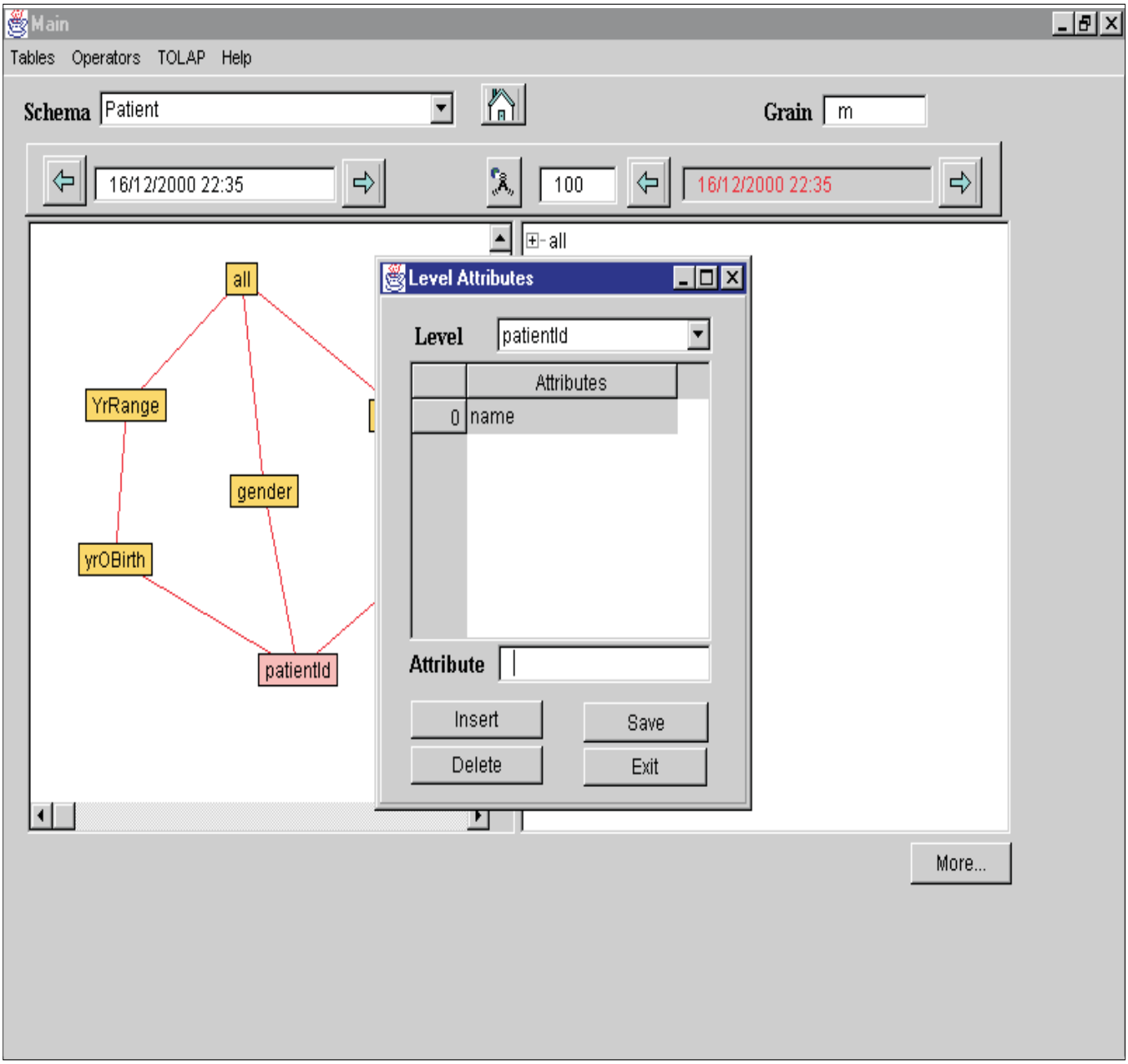

Figure 7.15: Attribute definition

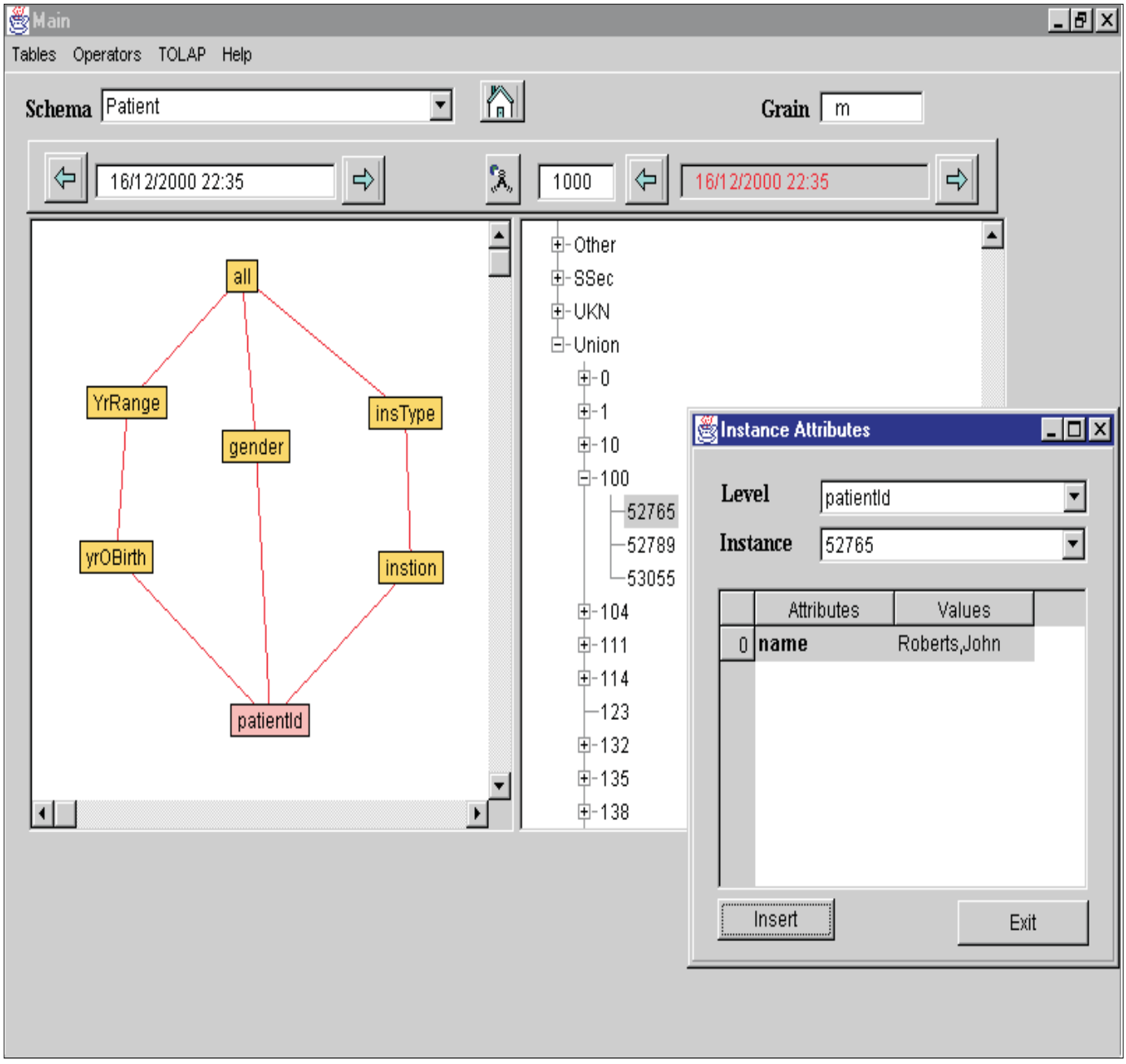

Figure 7.16: Attribute instances

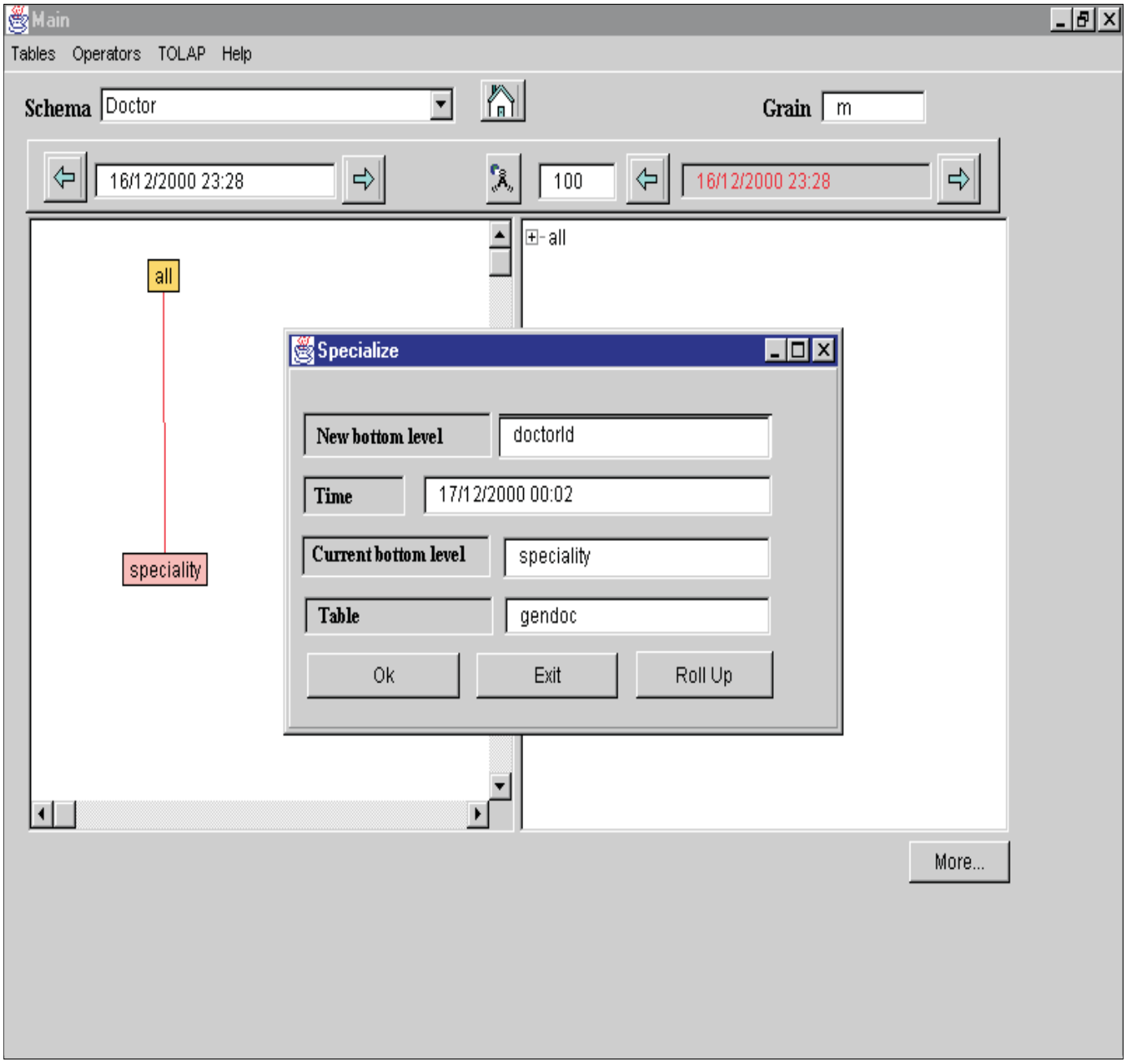

Figure 7.17: Specialization of level *doctorId*

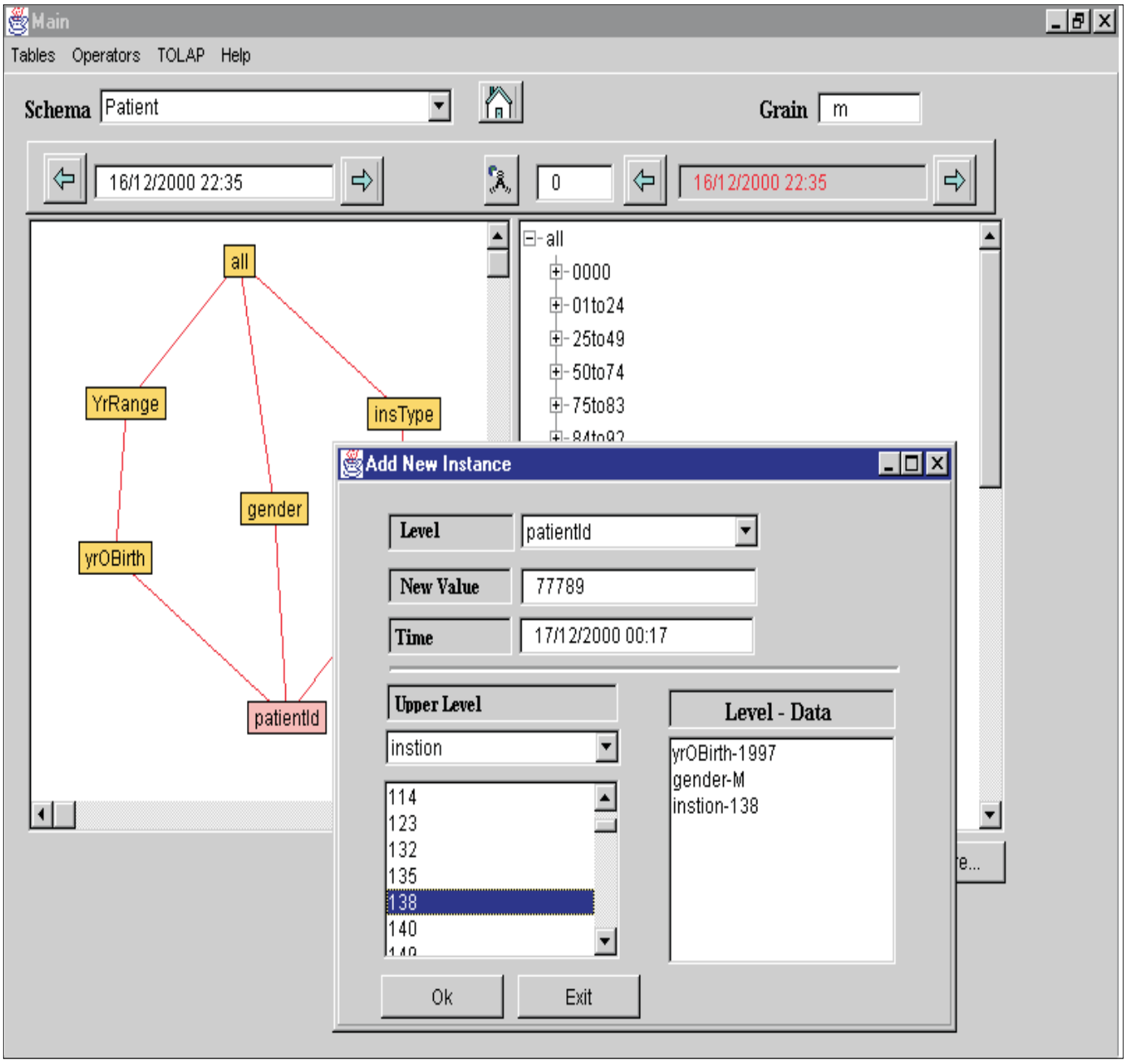

Figure 7.18: *AddInstance* operator

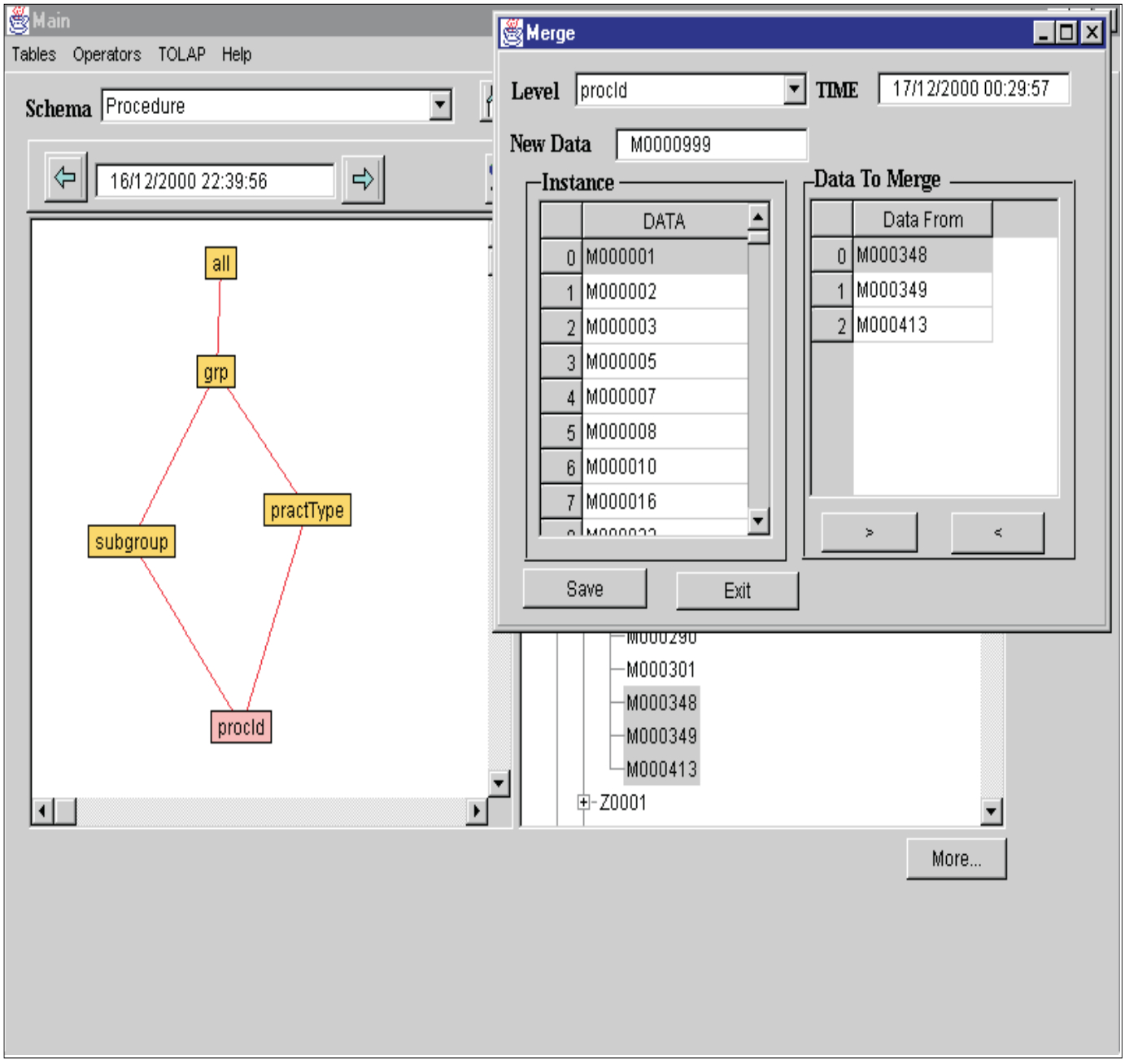

Figure 7.19: *Merge* operator

Chapter 8

Conclusion

In this thesis we have argued that dimensions in a data warehouse are not static entities, but are subject to updates, either at the schema or instance level. We studied the effect of these updates over a set of materialized views and developed a characterization of the possible dimension updates. We extended our approach to the temporal database framework and introduced a language which can express a class of OLAP queries.

8.1 Contributions

We summarize our contributions as follows:

We showed that supporting dimension updates is a desirable feature for an OLAP tool, in order to avoid constantly rebuilding dimensions from scratch when such an update occurs. We introduced a set of schema and instance update operators, and developed an algorithm to maintain materialized views under each one of these operators. Our algorithms in some cases outperform the well-known Summary-Delta Method [MQM97].

We implemented the update operators and an extension to the MDX language, Microsoft's standard for OLAP, following the OLE DB for OLAP standard.

We extended our proposal to the temporal database framework, introducing the *temporal multidimensional model,* a query language supporting it which we denoted *TOLAP*, and a set of temporal update operators. *TOLAP* accounts for schema evolution, and allows expressing complex queries at a high abstraction level. We also implemented the temporal model and developed a visualization tool which allows browsing the schema and instances of a dimension across time, creating and updating dimensions and fact tables, and editing and running *TOLAP* queries without leaving the environment.

All our implementations were tested using a real-life case study, a medical center in Buenos Aires.

8.2 Future Work

Our MDDLX proposal can be extended to support temporal updates, in a way such that MDDLX itself could allow temporal OLAP queries. The power of MDDLX can be increased allowing bulk updates over sets of elements. Moreover, we would like to add data cube bulk update support to MDDLX. View maintenance for complex instance dimension updates should be addressed in future versions of MDDLX, as well as data cubes with non-distributive aggregate functions.

TOLAP can be extended in order to allow the definition of constraints, which could be easily introduced within our visualization tool. Also, there is space for studying query optimization in *TOLAP*.

Another issue which deserves attention is adding update support to *TOLAP*, allowing bulk updates like "delete all customers which had no completed any transaction since 1998". Also, transactions in update expressions in *TOLAP* could be addressed. For example, the expression above could be followed by :" classify all customers which did not perform any transaction since 1999 as 'low priority' customers".

Appendix 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.

```
1.ALTER TIME DIMENSION Services.dayTimeDim
   GENERALIZE GRANULARITY DATETIME TO DATE

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

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

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

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

6.ALTER DIMENSION Services.Procedure
```

```
  GENERALIZE LEVEL procedureId
    TO LEVEL subgroup TYPE CHAR(5)
    USING ROLLUP FUNCTION data3gprocedureIdsubgroup

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

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

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

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

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

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

```
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
```

Bibliography

[AGS+96] R. Agrawal, A. Gupta, S. Sarawagi, P. Deshpande, S. Agarwal, J. Naughton, and R. Ramakrishnan. On the computation of multidimensional aggregates. In *Proceedings of the 22nd VLDB Conference*, Bombay, India, 1996.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[BSSJ98] R. Bliujute, S. Saltenis, G. Slivinskas, and G. Jensen. Systematic change management in dimensional data warehousing. *Time Center Technical Report TR-23*, 1998.

[BWJ98] R. Bettini, S. Wang, and S. Jajodia. Semantic assumptions and their use in databases. *IEEE Transactions on Knowledge and Data Engineering*, 1998.

[CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (on-line analytical processing) to user analysis : An it mandate. *White Paper. Arbor Software*, 1993.

[CKW89] W. Chen, M. Kifer, and D. S. Warren. Hilog as a platform for database language. In *Proceedings of the 2nd. International Workshop on Database Programming Languages*, pages 315–329, Oregon Coast,Oregon,USA, 1989.

[CM90] M. Consens and A.O. Mendelzon. Low complexity aggregation in Graphlog and Datalog. In *Proceedings of the 3rd International Conference on Database Theory, Lecture Notes in Computer Science n.470*, pages 379–394, 1990.

[CT97] L. Cabibbo and R. Torlone. Querying multidimensional databases. In *Proceedings of the 6th International Workshop on Database Programming Languages(DBPL'97)*, pages 253–269, East Park, Colorado, USA, 1997.

[FJS97] C. Faloutsos, H. Jagadish, and N. Sidiropoulos. Recovering information from summary data. In *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[GBLP97] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube : A relational operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery 1, pgs. 29-53*, 1997.

[GHQ95a] A. Gupta, V. Harinarayan, and D. Quass. Aggregate query processing in data warehousing environments. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[GHQ95b] A. Gupta, V. Harinarayan, and D. Quass. Generalized projections:a powerful approach to aggregation. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[GJM94] A. Gupta, H.V Jagadish, and I.S. Mumick. Data integration using self-maintainable views. *Technical Memorandum 113880-941101-32, ATT Bell Labs*, 1994.

[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.

[GMS93] A. Gupta, I.S. Mumick, and D Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, Washington D.C.,USA, 1993.

[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.

[Huy00] N. Huyn. Speeding up view maintenance using cheap filters at the warehouse. In *Proceedings of IEEE/ICDE'2000*, San Diego, USA, 2000.

[Inf96] Informix Corporation. *Informix OnLine Extended Parallel Server and Informix Universal Server : A New Generation of Decision-Support Indexing for Enterprisewide Data Warehouses*, 1996. White Paper.

[JLS99] H.V Jagadish, L.V.S Lakshmanan, and D. Srivastava. What can hierarchies do for data warehouses? In *Proceedings of the 25th VLDB Conference*, Edimburgh, Scotland, 1999.

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

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of ACM, p.699-717*, 1982.

[LSS93] L.V.S Lakshmanan, F. Sadri, and I.N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *Third International Conference on Deductive and Object-Oriented Databases(DOOD93), Springer-Verlag,LNCS-760*, 1993.

[LSS97] L.V.S Lakshmanan, F. Sadri, and I.N. Subramanian. Logic and algebraic languages for interoperability in multidatabase systems. *Journal of Logic Programming 33(2),pp.101-149*, 1997.

[LW95] D. Lomet and J. Widom. *IEEE Data Engineering Bulletin. Special Issue on Materialized Views and Data Warehousing*, June 1995.

[LW96] C. Li and S. Wang. A data model for supporting on-line analytical processing. In *Proceedings of the Conference on Information and Knowledge Management*, pages 81–88, 1996.

[LW97] L. Libkin and L. Wong. On the power of aggregation in relational query languages. In *Proceedings of the 6th International Workshop on Database Programming Languages(DBPL'97)*, pages 270–280, East Park, Colorado, USA, 1997.

[Mic] Microsoft Corporation. *OLEDB for OLAP 2.0 Design Specification.*

[Mic98] Microsoft Corporation. *OLEDB for OLAP Programmer's Reference (Internet Document http://www.microsoft.com/oledb/olap/spec)*, 1998.

[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.

[OLA97] OLAP Council. *OLAP Council White Paper*, 1997.

[Pil96] Pilot Software. *An introduction to OLAP*, 1996. White Paper.

[PJ99] T.B Pedersen and C. Jensen. Multidimensional data modeling for complex data. *Proceedings of IEEE/ICDE'99*, 1999.

[QGMW96] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data-warehousing. In *Parallel and Distributed Information Systems*, Miami, Florida, USA, 1996.

[Qua96] D. Quass. Maintenance expressions for views with aggregations. In *ACM Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, 1996.

[QW97] D. Quass and J. Widom. On-line warehouse view maintenance for batch updates. In *Proceedings of the ACM - SIGMOD Conference*, Tucson, Arizona, 1997.

[Sno95] Richard Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[Sta96] Stanford Technology Group. *Designing the Data Warehouse On Relational Data Warehouses*, 1996. White Paper.

[TBG+99] E. Thomsen, L. Baekgaard, D. Grossman, W. Liang, and S. Tolkin. Panel: Future directions in data warehousing. In *Proceedings of ACM DOLAP'99*, Kansas City, USA, 1999.

[Tom95] D. Toman. Point-based vs. interval-based temporal query languages. In *Proceedings of the ACM - PODS Conference*, 1995.

[Tom97] D. Toman. A point-based temporal extension to sql. In *Proceedings of DOOD'97*, Montreaux, Switzerland, 1997.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.

[Wid95] J Widom. Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, 1995.

[YW98] J. Yang and J. Widom. Maintaining temporal views over non-temporal information sources for data warehousing. In *Proceedings of the Sixth International Conference on Extending Database Technology*, Valencia, Spain, 1998.

[YW00] J. Yang and J. Widom. Temporal view self-maintenance in a warehousing environment. *To appear in Proceedings of the Seventh International Conference on Extending Database Technology*, 2000.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of ACM-SIGMOD Conference*, San Jose, California, 1995.