

A Temporal Query Language for OLAP: Implementation and a Case Study

Alejandro A. Vaisman¹ and Alberto O. Mendelzon²

¹ Universidad de Buenos Aires

avaisman@dc.uba.ar

² University of Toronto

mendel@db.toronto.edu

Abstract. Commercial OLAP systems usually treat OLAP dimensions as static entities. In practice, dimension updates are often necessary in order to adapt the multidimensional database to changing requirements. In earlier work we proposed a temporal multidimensional model and *TOLAP*, a query language supporting it, accounting for dimension updates and schema evolution at a high level of abstraction. In this paper we present our implementation of the model and the query language. We show how to translate a *TOLAP* program to SQL, and present a real-life case study, a medical center in Buenos Aires. We apply our implementation to this case study in order to show how our approach can address problems that occur in real situations and that current non-temporal commercial systems cannot deal with. We present results on query and dimension update performance, and briefly describe a visualization tool that allows editing and running *TOLAP* queries, performing dimension updates, and browsing dimensions across time.

1 Introduction

In models for OLAP (On Line Analytical Processing) [7,2,9], data is represented as a set of *dimensions* and *fact tables*. Dimensions are usually organized as hierarchies, supporting different levels of data aggregation. We argued in earlier work [5,6,10] that, although commercial OLAP tools largely do not support updates to dimension tables, these updates are likely to occur in many real-life situations. For instance, in Figure 1, a dimension *Geography* is represented as a hierarchy with levels *city*, *province*, *region*, *country*, and a distinguished level *All*. A business decision may allow regions to be spread across different countries(which is not allowed in the dimension of Figure 1). This change may be incorporated by deleting the edge joining the *region* and *country* levels, and adding a new edge from *region* to *All*.

Furthermore, thinking of a data warehouse as a materialized view of data located in multiple sources [14], it may happen that 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.

In earlier work [10] we showed that in an evolving OLAP scenario like the above, systems need temporal features to keep track of the different states of

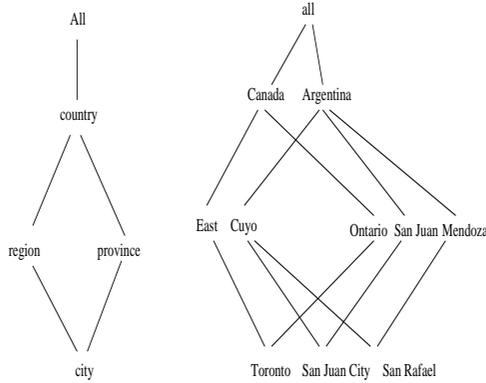


Fig. 1. A *Geography* dimension

a data warehouse throughout its lifespan. We introduced the *temporal multidimensional data model*, and a temporal OLAP query language called *TOLAP*. We showed that *TOLAP* expresses typical OLAP queries in a concise and elegant fashion. Moreover, *TOLAP* fully supports schema evolution and versioning, unlike the best-known temporal query languages such as TSQL2, which only supports schema versioning in a limited way ([12], p.29).

In this paper we present our implementation of the model and the query language. We show how to translate a *TOLAP* program to SQL and present a real-life case study, using data from a medical center in Buenos Aires, Argentina. We apply our implementation to this case study in order to show how our approach can address problems that occur in real situations and that current non-temporal commercial systems cannot deal with. We present results on query and dimension update performance, and briefly describe a visualization tool that allows editing and running *TOLAP* queries, performing dimension updates, and browsing dimensions across time.

Related Work

As mentioned above, in previous work we argued that in practice one encounters a wide range of possible dimension updates which the existing models fail to capture. Kimball analyzes this problem to some degree [7], introducing the concept of *slowly changing dimensions*, which partially covers updates to dimension instances. Kimball suggests some partial solutions, like timestamping dimension tuples with their validity intervals. This proposal neither takes schema versioning into account, nor considers complex dimension updates.

Work carried out at the *Time Center* at the University of Arizona [1] analyzes the performance of several SQL queries on three different implementations of an OLAP schema: “time series” fact tables, an “event” fact table, and dimensions timestamped in the way proposed by Kimball. This work was, to our knowledge, the first to suggest an approach to temporal OLAP. Our work went further by proposing a model and a query language to address temporal issues at a higher level of abstraction.

More recently, a multidimensional model for handling complex data considers the temporal aspect as a modeling issue [11], and addresses it in conjunction with other data modeling problems.

Paper Outline

The remainder of this paper is organized as follows: in Section 2 we review the temporal multidimensional data model. In Section 3 we introduce the case study, along with a review of *TOLAP* based on examples over this case. In Section 4 we describe the implementation of the system. In Section 5 we show how a translation of a *TOLAP* program to SQL proceeds. In Section 6 we present different tests performed over the case study, discussing dimension update performance, expressiveness and visualization capabilities. We conclude in Section 7.

2 The Temporal Multidimensional Model

Due to space limitations we will introduce the *Temporal Multidimensional Model* informally and by an example. We refer to our previous work for full details [10].

2.1 Temporal Dimensions

In what follows, we will consider time as discrete; that is, a point in the time-line, called a *time point*, will correspond to an integer.

A temporal dimension schema is a directed acyclic graph where each node represents a *level*, and each edge is labeled with the time interval within which the edge was/is valid. At any instant t , the graph is a partial order with a unique bottom level l_{inf} and a unique top level All .

Each dimension, at any time instant t , has an associated *instance* obtained by mapping each level to a set of *elements*. For each pair of levels connected by an edge, an instance defines a function ρ called *rollup*, that maps the elements of the source level to the elements of the destination level. The rollup function from level l_1 to level l_2 at instant t is denoted by $\rho[t]_{l_1}^{l_2}$. Moreover, dimension instances must satisfy the *consistency condition*: at every instant t in the dimension's lifespan, for every pair of paths from one level to another, composing the rollup functions along the two paths yields identical functions.

Notation: In the figures of this section, 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.

Example 1. Consider the dimension *Store* in Figure 2, with levels *storeId*, *storeType*, *city*, *region*, and *All*. The figure depicts the history of this dimension as follows: the initial levels were *storeId*, *city*, *region* and *All*. At time t_1 , level *storeType* was added above *storeId*. As there is only one possible top level, an edge is also added between *storeType* and *All*. This is called a *Generalization* [5].

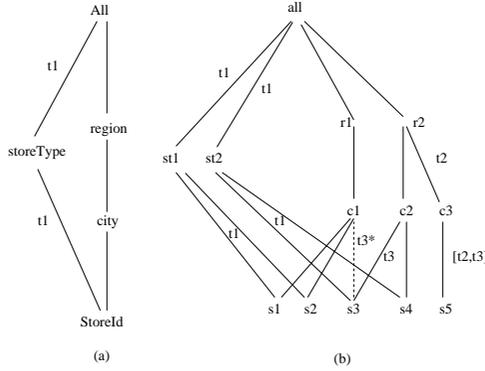


Fig. 2. (a) Temporal dimension schema (b) Temporal dimension instance

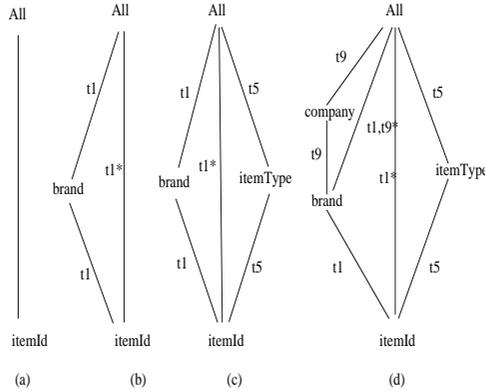


Fig. 3. A series of updates to dimension *Product*

Figure 2(b) shows a possible instance for the dimension *Store*. Here, store s_5 was valid between t_2 and t_3 , and store s_3 was moved from city c_1 to city c_2 at time t_3 ($\rho_{storeId}^{city}[t_3] = c_2$).

Figure 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, level *brand* is added to the dimension, although the initial state is not lost (Figure 3(b)). Later, the type of the item is inserted, with level name *itemType*. Finally, the *company* an item belongs to is also added above level *brand* (Figure 3(d)).

2.2 Temporal Fact Tables

Factual data is stored in temporal fact tables. We will assume that given a set of dimensions D , a temporal fact table has a column for each dimension in D , a column for a distinguished dimension *Measure*, and a column for the

Time dimension. A *temporal base fact table* is a temporal fact table such that its levels are the bottom levels of each one of the dimensions in D (plus *Measure* and *Time*). As these bottom levels may vary during the lifespan of the data warehouse, the schema of a base fact table can change. Keeping track of the different versions of the fact tables is called, like in temporal database literature, *schema versioning*. Note, however, that the attributes in any column of a fact table always belong to the same dimension.

A *Temporal Multidimensional Database* is a set of temporal dimensions and temporal fact tables.

Example 2. Given $D = \{Store, Product\}$ where dimensions *Store* and *Product* are the ones of Figures 2 and 3 respectively, the temporal base fact table associated to D would have levels $\{storeId, itemId, m, t\}$, where m is the measure and t is the time associated to a fact. If updates occur such that at time t_{12} , *brand* becomes the bottom level of dimension *Product*, the levels in the fact table will be $\{storeId, brand, m, t\}$.

In temporal databases, in general, instances are represented using *valid* time, this is, the time when the fact being recorded became valid. On the contrary, a database schema holds within a time interval that can be different from the one within which this schema was valid in reality. Changes in the real world are not reflected in the database until the database schema is updated. Schema versioning, thus, is often related with *transaction* time. In the temporal multidimensional model we are describing, things are different than above. An update to a dimension schema modifies the structure as well as the instance of the dimension. Figure 2 shows a dimension where schema and instance updates occurred. When *storeId* becomes generalized to *storeType*, the associated rollups must correspond to the instants in which they actually hold. Thus, we consider that temporal dimensions are represented by *valid* time. It is straightforward to extend the model for supporting temporal dimensions with valid and transaction times. However, at the moment, our implementation supports valid time for dimensions.

Fact table instances can be represented using valid and transaction times, because the model supports user-defined time dimensions. In our implementation, valid and transaction times for fact tables are supported through system-maintained and user-defined time attributes. In Subsection 2.3 we give an example of these kinds of time attributes. When a bottom level of a dimension is deleted or a level is added below it, the schema of the associated fact tables are modified, and new versions are created for them automatically. Thus, in our implementation, fact table versioning is represented using transaction time.

2.3 The Case Study: A Medical Data Warehouse

Throughout this paper we will refer to a real-life case study, a medical center in Argentina. We will use this example to illustrate the need for temporal management in OLAP. We used six months of data from medical procedures performed

on inpatients at the center. Each patient receives different services, including radiographies, electrocardiograms, medicine, disposable material, and so on. These services are denoted “Procedures”. Data was taken from different tables in the clinic’s operational database. Figure 4 shows the final state of the dimensions. We will explain in Section 6 how we simulated a temporal environment in which these dimensions were created at different times.

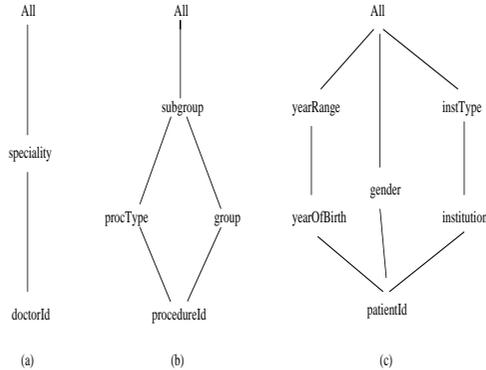


Fig. 4. Dimensions in the case study

Dimension *Procedure*, with bottom level *procedureId*, and levels *procedureType*, *subgroup* and *group*, describes the different procedures available to patients. Dimension *Patient*, with bottom level *patientId*, and levels *yearOfBirth* and *gender*, represents information about the person under treatment. Age intervals are represented by a dimension level called *yearRange*. Patients are also grouped according to their health insurance institution. Moreover, these institutions are further grouped into types (level *institutionType*), such as private institutions, labor unions, etc. Dimension *Doctor* gives information about the available doctors (identified by *doctorId*) and their *specialities* (a level above *doctorId*). There is also a user-defined dimension *Time*, with levels *day*, *week* and *month*.

A fact table holds data about procedures delivered to patients by a certain doctor on a certain date. We call this fact table *Services*, and its schema contains the bottom levels of the dimensions above, plus the measure. The fact table schema is : $Services(doctorID, procedureId, patientID, day, quantity, t)$, where t represents the system-maintained time dimension, and $quantity$ is the measure. For instance, a tuple $\langle doc1, 10.01.01, pat120, 10, 2, "10/10/2000" \rangle$ means that doctor “doc1” prescribed two special radiographies to patient “pat120” on day “10” which corresponds to October 10th, 2001. In this case study, the system-maintained and user-defined times are synchronized, meaning that there is a correspondence between the values of attributes “day” and “t” (e.g., day “11” corresponds to October 11st, 2000). This synchronization, however, is not mandatory in the model. Thus, both times can take independent values.

3 TOLAP

In this section we describe the temporal OLAP language *TOLAP* (*Temporal OLAP*) [10]. *TOLAP* combines some of the features of temporal query languages like TSQL2 or SQL/TP [12,13] with some of the high-order features of languages like HiLog or SchemaLog [4,8], in an OLAP setting. We highlight *TOLAP*'s main characteristics by means of examples, using the schema of the medical center described in the previous section .

TOLAP is a rule-based language; as in Datalog, each rule defines the predicate in the head using the literals in the body. It would be easy to give an SQL-like syntax to the language, but we find the rule-based syntax clearer and more concise.

We begin our description of *TOLAP* with queries not involving aggregates. A query returning the procedures delivered to patients affiliated to unions will be expressed in *TOLAP* as:

$$\text{SrvU}(\text{proc}, \text{pat}, \text{qty}, \text{t}) \leftarrow \text{Services}(\text{doc}, \text{proc}, \text{pat}, \text{day}, \text{qty}, \text{t}), \\ \text{pat}[\text{t}] \rightarrow \text{insType}: \text{'Union'};$$

This query returns a projection of the tuples in *Services* such that *pat* rolled up to *'Union'* when the service was delivered. Variable *pat* represents an element in the lowest level of the dimension *Patient*. A tuple in *Services* will contribute to the result if a patient *pat* was served by doctor *doc* on day *day* at time *t*, and *pat* was affiliated to an institution of type *'Union'* at the time of being treated. The expression *pat*[*t*] \rightarrow *insType*:*'Union'* is called a *rollup atom*, and *Services*(*doc*, *proc*, *pat*, *day*, *qty*, *t*) is a *fact atom*.

Queries with aggregates can readily be expressed in *TOLAP*. For example, consider the query: "total number of services per procedure type and week."

$$\text{SP}(\text{ty}, \text{w}, \text{SUM}(\text{qty})) \leftarrow \text{Services}(\text{doc}, \text{proc}, \text{pat}, \text{day}, \text{qty}, \text{t}), \\ \text{proc}[\text{t}] \rightarrow \text{procType}: \text{ty}, \text{day} \rightarrow \text{week}: \text{w};$$

Descriptive attributes of dimension levels can be used in *TOLAP* queries. Suppose we want the total number of services delivered by Dr. Roberts each week. In *TOLAP*:

$$\text{SB}(\text{w}, \text{SUM}(\text{qty})) \leftarrow \text{Services}(\text{doc}, \text{proc}, \text{pat}, \text{day}, \text{qty}, \text{t}), \text{day} \rightarrow \text{week}: \text{w}, \\ \text{doc}[\text{t}] \rightarrow \text{doctorId}: \text{dr}, \text{dr.name}: \text{'Roberts'};$$

The rollup function represented by the rollup atom *doc*[*t*] \rightarrow *doctorId*:*dr* will be the identity during the intervals in which *doctorId* is the dimension's bottom level. Thus, the atom *doc*[*t*] \rightarrow *doctorId*:*dr* allows defining the attribute *name* as an attribute of the level *doctorId*, no matter the location of this level within the dimension's hierarchy. The expression *dr.name*=*'Roberts'* is called a *descriptive atom*.

TOLAP also allows querying the metadata, supporting queries with no fact table in the body of the rules, which we call *metaqueries*. For example: “List the periods during which no heart surgery was available.”

$$\text{NoHeart}(t) \leftarrow \text{!Procedure:procId:proc}[t] \rightarrow \text{group:g}, \\ \text{g.desc} = \text{'Heart Surgery'}.$$

The term `!Procedure:procId:proc[t]` is a *negated rollup atom*. Note that we must specify the name of the dimension and level for variable `proc` in the atom `Procedure:procId:proc[t]→group:g`, because there is no fact table in the body of the rule to which `proc` could be bound.

As in Datalog, rules can be composed into programs. For example: “patients who were administered procedures belonging to types delivered more than one hundred times.” We first compute the total number of procedures by type, and use the result in the final query.

$$\text{PT}(\text{ty}, \text{COUNT}(\text{qty})) \leftarrow \text{Services}(\text{doc}, \text{proc}, \text{pat}, \text{day}, \text{qty}, t), \\ \text{proc}[t] \rightarrow \text{procType:ty}; \\ \text{Q}(\text{pat}) \leftarrow \text{Services}(\text{doc}, \text{proc}, \text{pat}, \text{day}, \text{qty}, t), \text{q} \geq 100, \\ \text{Ty100}(\text{ty}, \text{q}), \text{proc}[t] \rightarrow \text{procType}[t]:\text{ty};$$

The expression $\text{q} \geq 100$ is called a *constraint atom*.

4 Implementation

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

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 of the form $(\text{dimensionId}, \text{loLevel}, \text{toLevel}, \text{From}, \text{To})$ represents the structure of the dimensions in the data warehouse across time. A tuple in this relation means that in the dimension identified by *dimensionId*, level *loLevel* rolled up to level *toLevel* between instants *From* and *to*. For example, the schema of the data warehouse introduced in Section 2 will be represented by a relation with tuples $\langle D1, \text{procedureId}, \text{procedureType}, t_1, \text{Now} \rangle$, $\langle D1, \text{procedureId}, \text{group}, t_2, \text{Now} \rangle$, $\langle D1, \text{group}, \text{subgroup}, t_5, \text{Now} \rangle$, and so on.

We briefly discuss how instances of the dimensions are stored under each kind of representations.

Fixed Schema. Each dimension instance is represented by a relation with schema $(\text{loLevel}, \text{upLevel}, \text{loVal}, \text{upVal}, \text{From}, \text{To})$. Each tuple in this relation represents a rollup $\rho_{\text{loLevel}}^{\text{upLevel}}[(\text{From}, \text{To})(\text{loVal})] = \text{upVal}$. As an example, the instances of dimension *Procedure* of Section 3, would be represented by a relation with tuples of the form $\langle \text{procedureId}, \text{procedureType}, \text{pr}_1, \text{ty}_1, t_1, t_3 \rangle$,

$\langle procedureId, subgroup, pr_1, sg_1, t_2, t_3 \rangle$, and so on. Thus, as new levels are added to the dimension, new tuples are inserted, but the relation's schema does not change. For instance, adding a new level *priceRange* above *procedureId* implies adding tuples like $\langle procedureId, priceRange, pr_1, 200to300, t_6, Now \rangle$.

In this representation, dimension updates can be easily implemented, except updates that add or delete edges between levels, called *Relate* and *Unrelate* respectively [5,6]. These updates require self-joining the relation, in order to check consistency. Moreover, translation to SQL would be awkward, and the resulting SQL queries would require, again, self-joining temporal relations, as seen in the first SQL example below. In fact, self-joining temporal relations will be necessary each time the transitive closure of the rollup functions must be computed.

Non-fixed Schema. In this case there is also one relation for each dimension, but each dimension level is mapped to an attribute in this relation. A single tuple captures 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. For example, a relation representing dimension *Procedure* will have schema: $\{procedureId, procedureType, group, subgroup, From, To\}$.

This structure requires more complicated algorithms for supporting dimension updates. For instance, adding a dimension level above a given one (generalization), or below the bottom level (specialization), induces a schema update of the relation representing the dimension instance: a new column will be added to this 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. In the example above, a column *priceRange* would be added. However, the translation process is simpler, and the subsequent query performance better, because computing the transitive closure of the rollup functions reduces to a single relation scan.

To make these ideas more concrete, let us show how a *TOLAP* query is translated to SQL in each approach. (In Section 5 we give more details of this translation process.) Consider the query “total procedures by group.” This query reads in *TOLAP*:

```
GTot(g,SUM(qty)) ← Services(doc,proc,pat,day,qty,t), proc[t] → group:g;
```

Assuming that no schema update affected the fact table *Services*, the SQL equivalent of the *TOLAP* query above in the fixed schema approach will look like (for the sake of clarity we do not show here how time granularity is handled in the translation):

```
SELECT P1.upLevel,SUM(quantity)
FROM Services S, Procedure P, Procedure P1, Time T
WHERE
```

```

S.procedureId = P.loVal AND P.loLevel = 'procedureId' AND
P.upLevel = P1.loLevel AND P.upLevel = 'subgroup' AND
P1.upLevel = 'group' AND
S.time BETWEEN P.From AND P.To AND
S.time 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.group,SUM(quantity)
FROM Services S, Procedure P
WHERE
  S.procedureId = P.procedureId AND S.time BETWEEN P.From AND P.To
GROUP BY P.subgroup

```

Notice that computing the rollup from *procedureId* and *group* is straightforward, while in the first approach the self-join of the table *Procedure* is required.

These arguments led us to choose the non-fixed schema approach for our *TOLAP* implementation.

5 Translating *TOLAP* into SQL

The user interacts with the system through a visual interface. A dimension update module, a *TOLAP* parser and a *TOLAP* translator access the database through a JDBC driver. The *TOLAP* translator translates each atom into an SQL statement and builds the equivalent SQL query. We will show how a *TOLAP* rule of the form

$$Q(x,y,Ag(m)) \leftarrow F(x_i,y_j,m,t), x_i[t] \rightarrow l_i:x, y_j[Now] \rightarrow l_j:y, Dim:l:r[t] \rightarrow p:z;$$

is translated to SQL. We assume that x_i is bound to a dimension D_i and y_j to dimension D_j . Also, m represents the measure of the fact table F .

5.1 Translating *TOLAP* Atoms

- For each *rollup atom* like $x_i[t] \rightarrow l_i:x$, a selection clause is built as follows:
 $F.i = Dim_i.bottom$ AND $F.time$ BETWEEN $Dim_i.From$ AND $Dim_i.To$
 Dimension Dim_i is the table representing the dimension D_i . The first conjunct joins this dimension to the fact table on the attribute representing the bottom level. The actual name of the column $F.i$ is taken from the fact table's metadata. The second conjunct corresponds to the join between the fact table and the "Time" dimension.
- Each *time constant* is translated to a selection clause. The second rollup atom in the rule above will be translated as¹:
 $F.j = Dim_j.bottom$ AND $Dim_j.To = Now$

¹ In an actual implementation, *Now* can be replaced by *Sysdate()* or any function returning the current time.

- A rollup atom $\text{Dim}:l:r[t] \rightarrow p:z$, 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.

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


```
F.j = Dimj.bottom
```
- A *constraint atom* of the form $x \{<, =\} C$, where C is a constant term, 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 the negated atom $!(y_j[Now] \rightarrow l_1 : 'a')$, where 'a' represents a constant. This atom is converted into an SQL expression of the form:

```
NOT EXISTS(
  SELECT *
  FROM Dimj
  WHERE
    Dimj.To=Now AND Dimj.l1='a' AND F.j = Dimj.bottom)
```

where l_1 is the attribute representing level l_1 . A *negated descriptive atom* is translated analogously.

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

5.2 TOLAP Rules

So far we tackled the problem of translating each atom in a *TOLAP* rule separately. Now we will explain the translation of the whole rule. We put all the pieces of the WHERE clause together and use aggregation and projection for the rollup from the bottom levels of the dimensions D_i and D_j to the levels l_i and l_j in the head of the clause. Thus, the SQL query generated by the *TOLAP* query above will look like this:

```
SELECT Dimi.li, Dimj.lj, Ag(measure)
FROM F1, Dimi, Dimj
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 subindex represents the version of the fact table. So far there is only one version, as no schema update has occurred yet. However, we claimed that one of the main *TOLAP* features is the ability to deal with schema updates triggered by an specialization or a deletion of a bottom level. In the system's catalog, in a table describing fact table data, a new tuple is stored each time a dimension update affects a fact table in the data warehouse. Thus, given a fact table *F*, this fact table may have versions *F_1*, *F_2* and so on, with different schemas. Given a *TOLAP* rule *Γ* with a fact table *F* in the body, the SQL query *Q* equivalent to *Γ* will be such that $Q = Q_1 \cup Q_2 \cup \dots \cup Q_n$, where Q_i are the queries involving facts occurred in the intervals I_i in which each *F*.*i* holds. If the query includes an aggregate function, call it f_{AGG} , one more aggregation must be performed, in order to consider duplicates in each subquery. Thus, $Q_{AGG} = f_{AGG}(Q)$

Finally, *TOLAP* programs are compiled by translating each rule, creating temporary tables for each predicate in the head of a rule.

5.3 Join Elimination

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 is bound to the bottom level of a fact table in the body, or to a level that 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 3. Let us consider the query “total procedures by procedureId and doctor speciality”. Assume that the fact table *Services* was split into *Services_1* and *Services_2*, holding before and after a time instant t_5 respectively. In *Services_1*, the levels *procedureId* and *speciality* were the bottom levels corresponding to the dimensions *Procedure* and *Doctor*. In *Services_2*, these bottom levels were *procedureId* and *doctorId*. The query in *TOLAP*, and its SQL equivalent are:

```

PS(pro,sp,SUM(qty)) ← Services(doc,prac,pat,day,qty,t),
                    doc[t]→speciality:sp,
                    prac[t]→procedureId:pro;
SELECT procedureId,speciality,SUM(quantity)
FROM (
  SELECT procedureId,speciality, SUM(quantity)

```

```

FROM Services_1
GROUP BY procedureId, speciality
UNION ALL
SELECT procedureId, speciality, SUM(quantity)
FROM Services_2, Doctor
WHERE
Services_2.Time BETWEEN Doctor.From AND Doctor.To AND
Services_2.doctorId=Doctor.doctorId
GROUP BY procedureId, speciality )
GROUP BY procedureId, speciality

```

Notice that, as *speciality* and *procedureId* were the bottom levels of *Services_1*, no join is needed in the first subquery.

5.4 Subquery Pruning

If a *TOLAP* rule contains a constraint atom with a condition over time such that the lifespan of some 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 3, adding the constraint $t < d_6$ will prevent the first subquery from being generated.

5.5 Translating to TSQL2

Although at first sight TSQL2 might appear as the obvious target for translating *TOLAP* queries into an SQL-like query language, a more detailed analysis will show some drawbacks of this approach. In what follows we will assume that the reader is familiar with the design of TSQL2 [12].

TSQL2 has an *implicit* time data model. This feature, although useful for queries involving temporal joins, makes it difficult to express queries like the ones we deal with here. This is related to the problem of *lack of universality* of TQSL2, as studied by Chen and Zaniolo [3].

Although for some queries using TSQL2 would reduce or eliminate the need for complex expressions involving time, most of the time these expressions do not actually affect performance. Moreover, explicit time management makes *TOLAP* semantics easy to capture, whereas implicit time management turns this task difficult. For example, in TSQL2 the join of valid time relations returns another valid time relation. In order to avoid this behavior, different constructions must be used, for example the *SNAPSHOT* statement, which produces a non-temporal relation. Granularities must be explicitly handled using the *CAST* construct, often combined with *PERIOD* and other operators like *INTERSECT* or *CONTAINS*. Thus, although apparently simpler, the generated code contains many TSQL2 keywords, reducing portability and delivering no benefit, as *TOLAP* already hides all time management from the user.

For an example, consider the query: “total number of procedures by procedure subgroup and institution type” (we will be using this query for measuring performance later in Section 6):

```
Q(b,c,SUM(qty)) ← Services(doc,prac,pat,day,qty,t),
                  prac[t]→subgroup:c, pat[t]→institutionType:b;
```

For the sake of simplicity we will assume that there is only one fact table version. In TSQL2, this query would read (in bold we indicate the TSQL2 keywords):

```
SELECT SNAPSHOT COL_0, COL_1, SUM(SUM_2)
FROM (
  SELECT SNAPSHOT DIMENSION_PROCEDURE.subgroup AS COL_0,
         DIMENSION_PATIENT.instType AS COL_1, SUM(QUANTITY) AS SUM_2
  FROM SERVICES_1, DIMENSION_PATIENT, DIMENSION_PROCEDURE
  WHERE VALID(DIMENSION_PATIENT) CONTAINS VALID(SERVICES_1)
  AND SERVICES_1.PATIENTID = DIMENSION_PATIENT.PATIENTID AND
  VALID(DIMENSION_PROCEDURE) CONTAINS VALID(SERVICES_1)
  AND SERVICES_1.PROCID = DIMENSION_PROCEDURE.PROCID
  GROUP BY DIMENSION_PATIENT.INSTTYPE,
           DIMENSION_PROCEDURE.SUBGROUP)
GROUP BY COL_0, COL_1
```

As *TOLAP* allows considering different instants for rollup evaluation, even using TSQL2 as the target language cannot prevent explicit time management. Suppose the query above is replaced by:

```
Q(b,c,SUM(qty)) ← Services(doc,prac,pat,day,qty,t),
                  proc[Now]→subgroup:c,
                  pat['10/10/2000']→institutionType:b;
```

In this case, since the granularities of the arguments of the rollup functions and the dimensions differ, the translator must also generate the TSQL2 statements for casting these differences.

For the translation above, we assumed that the fact table was created as an *EVENT* relation, and the dimension tables as *STATE* relations. However, these assumptions carry further problems: as update semantics for TSQL2 is different than the semantics of dimension updates, ad-hoc modifications are needed to make the translation work. This would be much more expensive than generating the SQL expressions for explicit time management. Another problem would be the limited support provided by TSQL2 for schema versioning.

From the above we conclude that the benefits of translating *TOLAP* to TSQL2 are outweighed by the problems involved in adapting TSQL2’s semantics to the semantics of *TOLAP*.

6 The Medical Clinic Case Study

In this section we apply the temporal approach to the case study introduced in Section 2.3. Our temporal multidimensional data model lets us not only modify the dimensions on-line, but keep track of the history of the medical data warehouse. We will present a simulated scenario based on real data extracted from the system described in Section 2.3.

The objective of the study was to test the ability of the temporal approach and *TOLAP* to address user needs more fully than commercially available non-temporal OLAP tools. Using our implementation, we wanted to study: (a) performance, by measuring response time of dimension updates and *TOLAP* queries; (b) expressive power and abstraction, by posing queries which could not be easily expressed in non-temporal environments and comparing them with their SQL equivalent; and (c) visualization capabilities, through a graphical environment that supports browsing dimensions and fact tables across time, updating dimensions, and temporal querying of the warehouse.

6.1 Data Preparation

The testing scenario was prepared applying the following dimension updates. Dimension *Procedure* was created with bottom level *procedureId*. Subsequent operations performed using the graphic interface generalized this level to level *subgroup*. Level *procedureId* was then generalized to *procType*. Finally, *subgroup* was generalized to *group*, and *procType* related to *group*. (See Section 2.3 for explanations of these and all other dimension levels mentioned in this section). In the *Patient* dimension, the initial bottom level *patientId* represents information about the person under treatment. This level was then generalized to *yearOfBirth*, *gender* and *institution*, in that order. Levels *yearOfBirth* and *institution* were further generalized into *yearRange* and *institutionType* respectively. For dimension *Doctor*, in order to show how schema versioning is handled in *TOLAP*, we assumed that although facts were recorded at the *doctorId* level, this level was temporarily deleted, originating the second version of the fact table, called *Services.2*. Thus, during a short interval, the dimension's bottom level was *speciality*, later specialized into level *doctorId*, which triggered the creation of the third version of the fact table, denoted *Services.3*. Finally, a user-defined *Time* dimension was created, with granularity *day*, allowing expressing aggregates over time. The dimension's hierarchy is the following: *day* rolls up to *week* and *month*; *week* and *month* roll up to *All*. The three resulting fact table versions have the following schemas: (a) *Services.1*:{*procId*, *patientId*, *doctorId*, *day*, *quantity*, *tmp*}, where *tmp* represents the built-in time dimension, and *quantity* is the number of procedures delivered; (b) *Services.2*:{*procId*, *patientId*, *speciality*, *day*, *quantity*, *tmp*}; (c) *Services.3*:{*procId*, *patientId*, *doctorId*, *day*, *quantity*, *tmp*}. The fact tables were populated off-line.

The following table depicts the number of tuples in each table of the relational representation, after all the updates above occurred.

Table	# of tuples
Patient	16383
Procedure	11253
Doctor	822
Time	730
Services_1	26040
Services_2	12464
Services_3	17828

We also performed several updates at the instance level, simulating situations where new doctors were hired, others left, new procedures were created, or patients moved from one institution to another.

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 an ORACLE 8.04 database running on top of a Windows NT 4 (Service Pack 5) Operating System.

6.2 Discussion of Results

Performance. Figure 5 shows one of sets of *TOLAP* queries we ran. Query Q1 has three rollup atoms in the body, while query Q3 has only one. 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, see [10] for details).

```

Q1: Q(a,b,c,SUM(qty)) ← Services(doc,proc,pat,day,qty,t),
                        doc[t]→speciality:a, proc[t]→subgroup:c,
                        pat[t]→institutionType:b ;
Q2:  Q(b,c,SUM(qty)) ← Services(doc,proc,pat,day,qty,t),
                        proc[t]→subgroup:c, pat[t]→institutionType:b;
Q3:   Q(b,SUM(qty)) ← Services(doc,proc,pat,day,qty,t),
                        pat[t]→institutionType:b;

```

Fig. 5. 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(b,SUM(qty)) ← Services(doc,proc,pat,day,qty,t),
                pat[t]→institutionType:b, t < '02/13/2001' ;

```

The table below shows the query execution times for the three sets of queries described above. Each query was run 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 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.

	Query type	Q1	Q2	Q3
1	t	290 (977)	130 (361)	40 (4)
2	$t = \text{Now}$	250 (977)	110 (361)	30 (4)
3	$t < \text{"02/08/2001"}$	60 (289)	50 (95)	15 (4)
4	$t < \text{"02/13/2001"}$	140 (620)	125 (230)	20 (4)

We measured execution times for dimension updates in different ways. For instance, the table below shows the results of performing a sequence of dimension updates over dimension *Patient*. We include execution times for dimension updates over an equivalent non-temporal dimension *Patient* (created in *initial* mode, which does not account for the dimension's history). These results are indicated in the rightmost two columns of the following table.

# tuples in dim.	Update	Time (s)	# tuples (nt)	Time (s)(nt)
2727	create dimension	2	2727	2
2727	Gen. <i>patientId</i> to <i>gender</i>	20	2727	20
5452	Gen. <i>patientId</i> to <i>age</i>	30	2727	18
8179	Gen. <i>patientId</i> to <i>institution</i>	230	2727	20
10906	Gen. <i>age</i> to <i>ageRange</i>	30	2727	10
13633	Delete <i>gender</i>	15	2727	3
16356	Delete <i>institution</i>	20	2727	3
21808	Delete <i>age</i>	25	2727	3

Expressiveness. The queries below exemplify *TOLAP*'s expressive power applied to this case study. These queries cannot be expressed in a non-temporal model without ad-hoc design.

For example, suppose a user wants to analyze the doctors' workloads, in order to estimate future needs. The following query measures how 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 Dr. Roberts while Dr. Richards was not working for the clinic."

```
patRob(w,SUM(qty)) ← Services(doc,proc,pat,day,qty,t), day→week:w,
                    doc[t]→doctorId:d, d.name='Roberts',
                    !Doctor:doctorId:dd[t]→All:all, dd.name='Richards'.
```

Notice 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 by Dr. Roberts while both doctors were employed at the clinic.

```
patRob(w,SUM(qty)) ← Services(doc,proc,pat,day,qty,t), day→week:w,
                    doc[t]→doctorId:d, d.name='Roberts',
                    Doctor:doctorId:dd[t]→All:all, dd.name='Richards'.
```

The next query illustrates how to check patients who were served when they were affiliated to ‘MEDICUS’ and are currently affiliated to ‘OSDE’.

```
changePlan(pat) ← Services(doc,proc,pat,day,m,t),
                 pat[t]→institution:‘MEDICUS’,
                 pat[Now]→institution:‘OSDE’.
```

Visualization Capabilities. The third goal of our experiments consisted in exercising on this case study the graphical environment we developed to support the temporal multidimensional model. We used this environment to perform the dimension updates reported above and to browse the structures and instances of the dimensions across time, in order to test usefulness and performance of the tool. The graphic interface supports: (a) Browsing dimensions and instances across time, and seeing how they were hierarchically organized throughout their lifespan. (b) Performing dimension updates (c) Importing rollup functions from text files. (d) Browsing different versions of a fact table. (e) Sending *TOLAP* programs to the query engine, and displaying they results without leaving the environment, including the possibility to see the generated SQL query.

7 Conclusion and Future Work

We have described the implementation of *TOLAP*, a temporal OLAP query language introduced in a previous work [10]. We discussed two different relational representation alternatives, and gave details of the translation from *TOLAP* to SQL, or program, showing that a query that is concisely expressed in *TOLAP* takes many lines of complex SQL code. Finally, we tested our implementation on a real-life case study. Our preliminary results on query and dimension update performance suggest that *TOLAP* can be useful for overriding the limitations of non-temporal OLAP commercial tools.

Many research directions remain open. Query optimization in *TOLAP* is an obvious one. Also, *TOLAP* can be extended to allow the definition of integrity constraints, which could be easily introduced within our visualization tool. Another issue deserving attention is adding update support to *TOLAP*, allowing bulk updates like “delete all customers who have not completed any transaction since 1998.” Transactions in update expressions in *TOLAP* could be also addressed. For example, the expression above may be followed by: “classify all customers who did not perform any transaction since 1999 as ‘low priority customers’ ”.

Acknowledgements

The authors wish to thank Daniel Grippo and Claudio Tirelli, at the University of Buenos Aires for their help with the implementation of *TOLAP*.

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada, and by the FOMEC program for the University of Buenos Aires.

References

1. R. Bliujute, S. Saltenis, G. Slivinskas, and G. Jensen. Systematic change management in dimensional data warehousing. *Time Center Technical Report TR-23*, 1998.
2. L. Cabibbo and R. Torlone. A logical approach to multidimensional databases. In *EDBT'98: 6th International Conference on Extending Database Technology*, pages 253–269, Valencia, Spain, 1998.
3. C.X. Chen and C. Zaniolo. Universal temporal extensions for database languages. In *Proceedings of IEEE/ICDE'99*, Sydney, Australia, 1999.
4. 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.
5. C. Hurtado, A.O. Mendelzon, and A. Vaisman. Maintaining data cubes under dimension updates. *Proceedings of IEEE/ICDE'99*, 1999.
6. C. Hurtado, A.O. Mendelzon, and A. Vaisman. Updating OLAP dimensions. *Proceedings of ACM DOLAP'99*, 1999.
7. R. Kimball. *The Data Warehouse Toolkit*. J.Wiley and Sons, Inc, 1996.
8. 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.
9. W. Lehner. Modeling large OLAP scenarios. In *EDBT'98: 6th International Conference on Extending Database Technology*, Valencia, Spain, 1998.
10. A.O. Mendelzon and A. Vaisman. Temporal queries in OLAP. In *Proceedings of the 26th VLDB Conference*, Cairo, Egypt, 2000.
11. T.B Pedersen and C. Jensen. Multidimensional data modeling for complex data. *Proceedings of IEEE/ICDE'99*, 1999.
12. Richard Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.
13. D. Toman. A point-based temporal extension to sql. In *Proceedings of DOOD'97*, Montreaux, Switzerland, 1997.
14. J. Widom. Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, 1995.