On the Mismatch Between Multidimensionality and SQL

Oscar Romero and Alberto Abelló

Universitat Politècnica de Catalunya

Abstract. ROLAP tools are intended to ease information analysis and navigation through the whole Data Warehouse. These tools automatically generate a query according to the multidimensional operations performed by the end-user, using the relational database technology to implement multidimensionality and consequently, automatically translating multidimensional operations to SQL. In this paper, we consider this automatic translation process in detail and to do so, we present an exhaustive comparative (both theoretical and practical) between the multidimensional algebra and the relational one. Firstly, we discuss about the necessity of a multidimensional algebra with regard to the relational one and later, we thoroughly study those considerations to be made to guarantee the correctness of a cube-query (an SQL query making multidimensional sense). With this aim, we analyze the multidimensional algebra expressiveness with regard to SQL pointing out the features a query must satisfy to make multidimensional sense and we also focus on those problems that can arise in a cube-query due to SQL intrinsic restrictions. The SQL translation of an isolated operation does not represent a problem, but when mixing up the modifications brought about by a set of operations in a single cube-query, some conflicts derived from SQL could emerge depending on the operations involved. Therefore, if these problems are not detected and treated appropriately, the automatic translation can retrieve unexpected results.

1 Introduction

OLAP (*On-line Analytical Processing*) tools are intended to ease information analysis and navigation all through the organizational data previously integrated (homogenized, cleaned and filtered) in a huge repository of data, the *Data Warehouse*, used to extract relevant knowledge of the organization. Specifically, OLAP tools are conceived to exploit the Data Warehouse for analysis tasks based on *multidimensionality*, the main feature of these tools. The multidimensional conceptual view of data is distinguished by the *fact/dimension* dichotomy and it is characterized by representing data as if placed in an n-dimensional space, allowing us to easily understand and analyze data in terms of facts (the subjects of analysis) and dimensions showing the different points of view where a subject can be analyzed from. These characteristics are desirable since OLAP tools are aimed to enable analysts, managers, executives and in general those people

involved in decision making, to gain insight into data through fast queries and analytical tasks, allowing them to make better decisions and, in short, be more competitive.

More precisely, OLAP functionality is characterized by dynamic multidimensional analysis of consolidated enterprise data supporting end user analytical and navigational activities. "Navigation" means to interactively explore a data cube by drilling, rotating and screening, and as presented in [FBSV00] we consider "roll-up" (increase the aggregation level), "drill-down" (decrease the aggregation level), "screening and scoping" (select by means of a criterion evaluated against the data of a dimension), "slicing" (specify a single value for one or more members of a dimension) and "pivot" (reorient the multidimensional view), as the typical end user operations performed on data cubes. Some works, like [PJ01] and [ASS06], add "drill-across" (combine data from cubes sharing one or more dimensions) to these basic operations.

In general, building up a *Data Warehousing System* (a Data Warehouse all together with its exploitation tools) is never an easy job, raising up some interesting challenges. Data must be gathered and assembled from various and possibly heterogeneous sources in order to gain a single and detailed view of the organization (the Data Warehouse) that later, must be properly managed and exploited. One of these challenges focus on modeling multidimensionality. Despite lacking of an standard multidimensional model like in the relational model, lots of efforts have been devoted to multidimensional modeling and several models have been developed. Consequently, we can nowadays design a multidimensional conceptual schema, create it physically and later, exploit it through the model algebra or calculus (implemented in the exploitation tools).

When implementing (i.e. creating it physically) our conceptual schema, and in general, the Data Warehousing System, over a DBMS (*Database Management System*) there are two main trends: to use the relational technology or an adhoc one, giving rise, respectively, to what are known as *ROLAP* (*Relational On-line Analytical Processing*) and *MOLAP* (*Multidimensional On-line Analytical Processing*) tools. ROLAP tools map the multidimensional model over the relational one, allowing them to make profit of a well-known and established technology and being, nowadays, the most frequent way to implement a Data Warehousing System. Specifically, [KRTR98] presents how a Data Warehouse should be implemented over a RDBMS (*Relational Database Management System*) and how to retrieve data from it. But, unfortunately, because of the lack of a multidimensional algebra accepted as reference point, there is not yet a widely accepted trend to translate the multidimensional algebra operators to SQL.

In the last years several algebras have already been proposed but some of them do not directly map to SQL and, in general, none of them offers the translation of its operators to SQL (rather they propose alternatives to SQL and the relational algebra). In fact, there are some models proposing alternatives to SQL arguing that RDBMS are not well suited for multidimensional purposes. One of this alternatives proposed is the MDX (*Multidimensional Expressions*) language ([Mic]). Developed by Microsoft for multidimensional tasks, MDX provides a

rich and powerful language to handle multidimensional data based on the SQL syntax, even though it is not an extension of SQL. However, MDX queries are static and therefore, we can not navigate all over the multidimensional data concatenating operators like an algebra does. MDX does not fit multidimensionality necessities better than SQL either, since these kind of tools aim for a language easing the user navigation and analysis of data from a user friendly perspective, and MDX, like SQL, are not easy to understand for a non-expert user. Moreover, it is also a declarative language and therefore, it needs to be translated to a procedural language; like SQL must be translated to the relational algebra. Nevertheless, MDX could be used as an intermediate language where to translate the end-user multidimensional operators to, but in that case, SQL (used in ROLAP tools, nowadays, the most extended way to implement a data warehouse system) may fit better since it is standard, well-known and its importance in the market is unquestionable.

In this paper, we focus on the automatic translation of the multidimensional algebra to SQL, and eventually to the relational algebra, that a ROLAP tool must implicitly perform. To do so, we present an exhaustive comparative (both theoretical and practical) between the multidimensional algebra and the relational one. However, since, unfortunately, we can not benefit from an standard multidimensional model, section 2 presents $\mathbf{YAM}^2$ as the multidimensional framework to be used as reference all over this paper, allowing us to compare both approaches.

Section 3 starts our discussion justifying why the relational algebra does not directly fit properly to multidimensionality by means of a conceptual comparison between both algebras. Later, we present a comparative between those multidimensional algebras introduced in the literature in the last ten years with regard to our framework. We conclude this section discussing about the necessity of a multidimensional algebra, the current state of the art and the suitability of $\mathbf{YAM}^2$ as our multidimensional framework along this paper.

Next, we present a thorough (practical) comparison between both models according to the automatic translation a ROLAP tool would perform. Thus, we go one step beyond presenting a logical comparative where the multidimensional algebra faces SQL as performed in a ROLAP tool translation process. Section 4 presents, in general terms, how to translate our framework (focusing on the multidimensional algebra) to SQL, and for each multidimensional operator we present its isolated translation to SQL. Section 5 studies in detail those additional considerations to be made to also guarantee the semantic correctness of a *cube-query* (an SQL following the multidimensional SQL pattern). With this aim, we first analyze the multidimensional algebra expressiveness with regard to SQL pointing out the features a query must satisfy to make multidimensional sense (i.e. to be a valid cube-query). Furthermore, we introduce a detailed algorithm based on those features spotted in this section to automatically infer if a correct SQL query is a valid cube-query. Finally, section 6 focus on those problems that can arise in a cube-query due to SQL intrinsic restrictions. ROLAP tools automatically generate a cube-query according to the multidimensional op-

erations performed by the user. The SQL translation of an isolated operation does not represent a problem and can be easily obtained as presented in section 4, but when mixing up the modifications brought about by a set of operations in a single cube-query, some conflicts derived from SQL could emerge depending on the operations involved. Therefore, if these problems are not detected and treated appropriately, the automatic translation can retrieve unexpected results. Consequently, we also analyze how to solve or minimize their impact. Finally, we present our conclusions and future work in section 7.

2 Our Framework: YAM^2

Due to the lack of an standard multidimensional model, and hence, the lack of a common notation, to carry out our work we need a reference framework in which to translate and compare the multidimensional algebras presented in the literature. Moreover, we will also need to compare that framework to the relational algebra since, nowadays, ROLAP tools are the most widely spread approach to model multidimensionality and therefore, multidimensional queries are being translated to SQL and (eventually) to the relational algebra. Conversely, a comparison among all those different algebras would be rather difficult.

In this section we present **YAM^2** (*Yet Another Multidimensional Model*) [ASS06], to be used as our reference multidimensional framework along this paper. Therefore, we present **YAM^2** data structure, its algebra and its integrity constraints to define concisely and univocally the multidimensional concepts as well as to provide a common notation all through this paper. From here on, **YAM^2** concepts will be **bold faced** for the sake of comprehension.

2.1 Data Structure

YAM^2 data structure was introduced in [ASS06] as an extension of UML. On one hand, a **Dimension** (subclass of UML *Classifier*) contains a hierarchy of **Levels** (subclass of UML *Class*) representing different granularities (or levels of detail) to study data, and a **Level** contains **Descriptors** (subclass of UML *Attribute*). We differentiate between identifier **Descriptors** and non-identifier. The first univocally identify each instance of a **Level**, in a role similar to the "primary key" in the relational model. On the other hand, a **Fact** (subclass of UML *Classifier*) contains **Cells** (subclass of UML *Class*) which contain **Measures** (subclass of UML *Attribute*). One **Cell** represents those individual **cells** of the same granularity that show data regarding the same **Fact** (i.e. a **Cell** is a "Class" and **cells** are its instances). Specifically, a **Cell** of data is related to one **Level** for each of its associated **Dimension** of analysis. We call a **Base** to those minimal set of **Levels** identifying univocally a **Cell**, similar to the "primary key" concept in the relational model. Therefore, **Dimension Levels** determine the multidimensional space where each **cell** is placed. A set of **cells** placed in the multidimensional space with regard to the **Base** is called a **Cube**. Finally, one **Fact** and several **Dimension**s to analyze it give rise to a **Star**. As discussed in

[ASS06], we consider quite important to be able to relate different **Star**s not only sharing dimensions but defining semantic relationships at design time between them like UML *Generalization*, *Association*, *Derivation* or *Flow*; some of them already considered in other conceptual models as [TPGS01] and [TBC99].

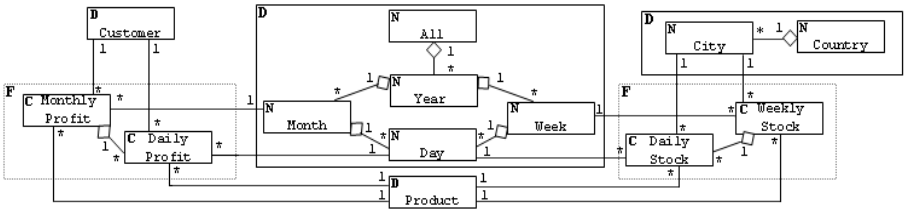

Fig. 1. Example of a multi-star schema

For instance, in figure 1 we find two **Fact**s containing two **Cell**s each one (the **Fact** `profit` containing the `Daily Profit` and `Monthly Profit` **Cells**, and the **Fact** `stock` containing the `Daily Stock` and `Weekly Stock` **Cells**). Both **Facts** are related to its **Dimensions** of analysis, and in this case, they are sharing two of them; the `Time` (showing explicitly its **Levels** hierarchy) and `Product` **Dimension**. Note the special `All` **Level** depicted in the `Time` **Dimension** hierarchy. This **Level** contains a unique instance representing the whole elementary instances of the **Dimension**; that is, represents the whole **Dimension**, and it must always placed in the top of the hierarchy. Also notice the importance of consolidation of data in the multidimensional model, where a value in a single **cell** may represent an aggregated measure computed from more specific data at some lower **Level** of the same **Dimension**. For instance, the `Monthly profit` data may have been consolidated as the sum of each month `Daily profit` disaggregated data.

Once we have presented our framework data structure notation, we can emphasize how these concepts should be implemented over a RDBMS. [KRTR98] shows how a **Star** should be implemented on RDBMS through a *star* or a *snowflake* schema. The star schema consists of one table for the **Fact** and one denormalized table for every **Dimension** with the latter being pointed by "foreign keys" (FK) from the "fact table", which compose its "primary key" (PK). The normalized version of a star schema is a snowflake schema, getting a table for each **Level** with a FK pointing to each of its parents in the **Dimension** hierarchy. Nevertheless, both approaches can be conceptually generalized into a more generic one consisting in partially normalizing the **Dimension** tables according to our needs. Completely normalizing each **Dimension** we get a snowflake schema and not normalizing them at all results in a star schema. We choose this generic approach as we consider, like in [MK00], a **Fact** can contain not just one but several materialized **Cell**s ("Cell tables"). So that, each **Level** related to a materialized **Cell** must also be materialized as a table since a FK (each FK in

the **Cell** pointing to **Level**s related to it) must be related to a PK, or at least, to a "unique" table field. If a certain **Level** is only related to non materialized **Cells** we can denormalize it. Semantic relationships are always translated as FK pointing to a "candidate key" (CK) without considering its semantics. In figure 1, we have decided to materialize the four **Cell**s stated explicitly (i.e. `Daily Stock`, `Weekly Stock`, `Daily Profit` and `Monthly Profit`). Hence, those **Level**s directly related to them will be materialized, but, for instance, `Year` **Level** will not since no materialized **Cell** points to it.

2.2 Multidimensional Algebra

In this section we present **YAM**^{2} operations introduced in detail in [ASS03], intended to manipulate **Cube**s.

- **Selection**: By means of a logic clause C over a **Descriptor**, this operation allows to choose the subset of points of interest out of the whole n-dimensional space.
- **Roll-up**: It groups **cell**s in the **Cube** based on an aggregation hierarchy. This operation modifies the granularity of data by means of a many-to-one relationship which relates instances of two **Level**s in the same **Dimension**, corresponding to a part-whole relationship.
- **ChangeBase**: This operation reallocates exactly the same instances of a **Cube** into a new n-dimensional space with exactly the same number of points, by means of a one-to-one relationship. Actually, it allows two different kinds of changes in the space **Base**. We can just rearrange the multidimensional space by reordering the **Levels** (this would be equivalent to the "pivot" operation), or, if exists more than one set of **Dimensions** identifying the **cells** (i.e. there are alternative **Bases**), by replacing the current **Base** by one of the alternatives ones.
- **Drill-across**: This operation changes the subject of analysis of the **Cube** by means of a one-to-one relationship. The n-dimensional space remains exactly the same, only the **cell**s placed in it change. Like in the **ChangeBase** operation, semantic relations rise new possibilities as presented in [ASS03].
- **Projection**: It selects a subset of **Measure**s from those available in the **Cube**.
- **Union**: It unites two **Cube**s containing the same **Cells** if both are defined over the same n-dimensional space. Same considerations could be done to define **Difference** and **Intersection**, just changing the logical operator applied between **Cubes** (the "OR", "AND NOT" an "AND" operators respectively). Notice, however, **Intersection** can be derived from **Difference** and therefore, it is not necessary in a minimal set of operations. From here on, we will just talk about **Union** for the sake of briefness, despite any consideration related to it can also be easily extended to **Difference** and **Intersection** as presented.

The algebra composed by these operations is "closed" (applied to a **Cube**, the result of all operations is another **Cube**), "complete" (any valid **Cube** can

be computed as the combination of a finite set of operations applied to the appropriate **Cell**) and "minimal" (none can be expressed in terms of others, nor can any operation be dropped without affecting its functionality). Therefore, other operations can be derived by sequences of these operations. This is the case of **Slice** (which reduces the dimensionality of the original **Cube** by fixing a point in a **Dimension**) by means of **Selection** and **ChangeBase** operations. For instance, referring to figure 1, we can **Slice** `Weekly Stock` fixing `Place` **Dimension** to a concrete value (i.e. `Barcelona)` by means of a **Selection**, and being `Time` $\times$ `Product` $\times$ 1 the current space **Base**. At this time, we can change the space base to `Time` $\times$ `Product` through a **ChangeBase** without losing **cell**s. About **Drill-down** (i.e. the inverse of **Roll-up**), as argued in [HS97], it can only be applied if we previously performed a **Roll-up** and did not lose the correspondences between **cell**s. Losing correspondences can happen due to extra navigation between **Cube**s (through **Drill-across** or **ChangeBase**) resulting that we do not have data in a lower aggregation **Level** for the target **Cube**.

2.3 Integrity Constraints

This section presents the multidimensional model integrity restrictions to be guaranteed at every moment. Integrity constraints pay attention to two important multidimensional aspects; placement of data in a multidimensional space and summarizability of data.

In one hand, first integrity constraint enforces us to identify each **Cell** instance by means of those **Levels** related to it. **Cells** (i.e. data) is placed in a n-multidimensional space conformed by its n **Dimensions** of analysis. For each one of its **Dimensions**, a **Cell** will be related to one **Level** of the **Dimension** hierarchy. Therefore:

- Every minimal set of **Levels** completely identifying a **Cell** is called a **Base**.

Notice the **Base** concept is similar to the "key" concept in the relational model, and it enforces us to keep, in every **Cube**, a functional dependency between **cells** and **Levels**. On the other hand, we present here the three necessary conditions (intuitively also sufficient) introduced in [LS97] to warrant a correct data summarization:

- **Disjointness:** *Sets of* **cell***s at an specific* **Level** *to be aggregated must be disjoint.*
- **Completeness:** *Every* **cell** *at a certain* **Level** *must be aggregated in some parent* **Level**.
- **Compatibility**: **Dimension**, *kind of measure aggregated and the aggregation function must be compatible.* Compatibility must be satisfied since certain functions are incompatible with some **Dimension**s and kind of measures. For instance, we can not aggregate `Stock` over `Time` **Dimension** by means of sum, as some repeated values would be counted.

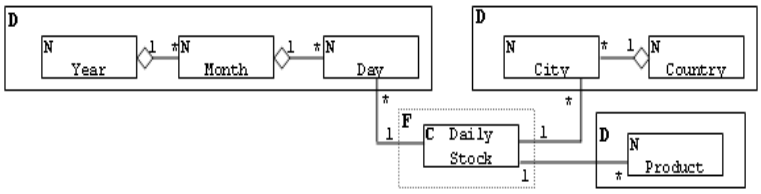

Fig. 2. Schema of a multidimensional Cube

When aggregating data we have to assure these conditions to avoid summarizability anomalies. If not, we will face duplicated values or find that some measurements at an aggregation **Level** cannot be used to obtain data at higher aggregation **Level**s, forcing us to go to finer granularities, maybe to the "atomic **Level**" (lowest **Level** in a **Dimension** hierarchy which is always materialized), to obtain the source data for the calculation.

3 Comparison of Algebras

In this section we have two main objectives. On one hand, we justify the necessity of a multidimensional algebra and why the relational one does not directly fit to multidimensionality needs. On the other hand, we justify our framework choice comparing it with all the multidimensional algebras presented up to now in the literature. Consequently, it also reveals the current state of the art. Finally, we discuss about the results presented along this section.

3.1 Multidimensional Algebra Vs. Relational Algebra

This section aims to justify the necessity of a semantic layer (the multidimensional algebra) on the top of the RDBMS (i.e. the relational algebra). Despite we believe ROLAP tools are the best choice to implement multidimensionality, we present, by means of a conceptual comparative between the multidimensional and the relational algebra operators, why the relational algebra (and therefore SQL) does not directly fit properly to multidimensionality. Furthermore, we emphasize in those restrictions and considerations needed to be made over the relational algebra with regard to multidimensionality.

In this comparative we consider the relational algebra presented in [Cod72]. Thus, we consider "Selection" (σ), "Projection" (π), "Union" ($\cup$), "Difference" ($-$) and "Natural Join" ($\bowtie$) as the relational algebra operators. As remark, we talk about "Natural Join", or simply "Join", instead of the "Cartesian Product" (the one presented in [Cod72] and where "Join" can be derived from) since a "Cartesian Product" without restrictions is meaningless in the multidimensional model, as discussed in [RA05]. Moreover, we do not include the "rename" operator, not included in [Cod72] but widely accepted later. This is because we are

focusing on handling and manipulating data and "rename" can be considered a meta-operator more than an operator by itself.

For the sake of comprehension, since we focus on a conceptual comparison, and to avoid messing results with considerations about the Data Warehouse implementation, we will consider, without loss of generality, that each multidimensional **Cube** is implemented as a single relation (i.e. a denormalized relational table). So that, considering the **Cube** depicted in figure 2 (extracted from figure 1) we would get the following relation: {`City`, `Day`, `Product`, `Daily Stock`, `Country`, `Month`, `Year`}. Being the underlined fields the multidimensional **Base** and therefore, the relation "primary key". Along this section, we will refer to this kind of denormalized relation as the *multidimensional table*.

Prior to present our results, just remind section 4 presents how each multidimensional operator should be translated to SQL, helping the reader to better understand this section.

Table 1 summarizes the mapping between both algebras operators. Notice we are considering the "group by" and "aggregation" as relational operators, and both will be justified consequently below. Since *multidimensional tables* contain (1) identifier fields (i.e. identifier **Descriptors** -see section 2.1-) identifying data -for instance: `City`, `Day` and `Product` in the above example-; (2) numerical fields: -`Daily Stock`-, representing multidimensional data (i.e. **Measures**) and (3) descriptive fields: -`Country`, `Month` and `Year`- (i.e. non-identifier **Descriptors**), we use the following notation in the table: $\checkmark_{Measures}$ if the multidimensional operator is equivalent to the relational one but it can be only applied over relation fields representing **Measures**, $\checkmark_{Descs}$ if the multidimensional operator must be applied over **Descriptors** fields and finally, $\checkmark_{Descs_{id}}$ if it can be only applied over identifier **Descriptors** fields. Consequently, a ✓ without restrictions means both operators are equivalent, without additional restrictions. If the translation of a multidimensional operator combines more than one relational operator, the subscript + is added.

YAM^2 Operator		"Selection"	"Projection"	"Join"	"Union"	"Group by"	"Aggregation"
Selection		$\checkmark_{Descs}$					
Projection			$\checkmark_{Measures}$				
Roll-up						$\checkmark_{Descs_{id}}+$	$\checkmark_{Measures}+$
Drill-across			$\checkmark_{Descs_{id}}+$	$\checkmark_{Descs_{id}}+$			
changeBase	**Add Dim.**			$\checkmark_{Descs_{id}}$			
	Remove Dim.		$\checkmark_{Descs_{id}}$				
	Alt. Base		$\checkmark_{Descs_{id}}+$	$\checkmark_{Descs_{id}}+$			
Union					✓		

Table 1. Comparative table between the relational and the multidimensional algebras.

– The multidimensional **Selection** operator is equivalent to a restricted relational "Selection". It can be only applied over **Descriptors** and then, it is equivalent to restrict the relational "Selection" just over **Level** data. According to our notation, we express the multidimensional **Selection** in terms of the relational algebra as $\sigma_{Descriptors}$.

– Similarly, the multidimensional **Projection** operator is equivalent to the relational one restricted to **Measures**; that is, specific **Cell** data. In terms of the relational algebra we could express it as $\pi_{Measures}$.
– OLAP tools emphasize on flexible data grouping and efficient aggregation evaluation over groups and it is the multidimensional **Roll-up** operator the one aimed to provide us with powerful grouping and aggregation of data. In order to support it, we need to extend the relational algebra to provide grouping and aggregation mechanisms. This topic have already been studied and previous works like [Klu82], [LW96] and [Lar99] have already presented extensions of the relational algebra to what is also called the *grouping algebra*. All of them introduce two new operators, and following the [Lar99] grouping algebra, we will refer to them as the "group by" and the "aggregation" operators.
The "group by" operator presented allows us to group data and apply a simple addition, counting or maximization of a collection of domain values (like it has been typically introduced, tightly connected to aggregation), and it also allows to perform relational computations on groups, even without applying aggregation. Moreover, it supports *nested groupings*, extending it with respect to more than one relational argument, fulfilling the OLAP necessities about grouping. The syntaxis introduced is the following: **group** $r_1, ..., r_n$ **by** $\mathcal{X}$ **do** e. Where $r_1, ..., r_n$ are relational names, $\mathcal{X}$ a subset of attribute names and e is a relational expression (even another "group by" expression). For instance, if we would like to **group** data **by** the `Product` field in the *multidimensional table* depicted in table 2 we would obtain: {{[Scarf, Spain, Barcelona, 10, 1, 1, 2006],[Scarf, Italy, Rome, 9, 1, 1, 2006]},{[T-shirt, Spain, Barcelona, 7, 1, 1, 2006], [T-shirt, Italy, Rome, 50, 1, 1, 2006]}, {[Socks, Spain, Barcelona, 30, 1, 1, 2006]}}.

Country	City	Product	Sales	Day	Month	Year
Spain	Barcelona	Scarf	10	1	1	2006
Italy	Rome	Scarf	9	1	1	2006
Spain	Barcelona	T-Shirt	7	1	1	2006
Italy	Rome	T-Shirt	50	1	1	2006
Spain	Barcelona	Socks	30	1	1	2006

Table 2. Implementation of figure 2 through a denormalized table (a *multidimensional table*).

Finally, the "aggregation" operator computes the aggregation of a given attribute over a given nested relation. Hence, it could be any of the usual aggregation functions, like SUM, COUNT, MAX, MIN, AVG, etc. For instance, $MAX_{Sales}(\mathtt{T})$, being `T` table 2, would evaluate to `50`.
In terms of this grouping algebra, a **Roll-up** operator consists of a proper "group by" operation along with an "aggregation" of data. Following our

example, **group** `T` **by** `Product` **do** SUM(`Sales`) would calculate the `Sales` sum per product; evaluating to {[Scarf,19],[T-Shirt,57],[Socks,30]}. Keep in mind this operation must perform a proper aggregation of data if we want it to be consistent.

– A consistent **Drill-across** typically consists on a "Join" between two *multidimensional tables* sharing the same multidimensional space. Notice that to "Join" both tables it must be performed over their common **Level** identifiers that must univocally identify each **cell** in the multidimensional space (the **Cube Base**). Moreover, once "joined", we must "project" out the columns in the *multidimensional table drill-acrossed* to, except for its **Measures**. Formally, Let $\mathcal{A}$ and $\mathcal{B}$ be the *multidimensional tables* implementing, respectively, the origin and the destination **Cells** involved. In the relational algebra it can be expressed as:

$$\pi_{Descriptors_{\mathcal{A}},Measures_{\mathcal{A}},Measures_{\mathcal{B}}}(\mathcal{A} \bowtie \mathcal{B})$$

– As stated in section 2, **changeBase** allows us to rearrange our current multidimensional space either by changing to an alternative **Base** (adding / removing a **Dimension** or replacing **Dimensions**) or reordering the space (i.e. "pivoting").
When changing to an alternative **Base** we must assure it does not affect the functional dependency of data with regard to the **Cube Base**. Hence:

- To add a **Dimension** it must be done through its `All` **Level** or fixing just one value at any other **Level** by means of a **Selection**, to not lose **cells** (i.e. representing the whole **Dimension** as a unique instance as discussed in 2.2). Therefore, in the relational algebra adding a **Dimension** is achieved through a "cartesian product" between the *multidimensional table* and the **Dimension** table (that would contain a unique instance). Specifically, being $\mathcal{C}$ the initial *multidimensional table* and $\mathcal{D}$ the relational table implementing the added **Dimension**, it can be expressed as:
$$\mathcal{C} \times \mathcal{D}, \quad where\ |\mathcal{D}| = 1$$
- To remove a **Dimension** it is just the opposite, and we need to get rid of the proper **Level** identifier projecting it out in the *multidimensional table*.
- To change the set of **Dimensions** identifying each **cell**, that is, choosing an alternative **Base** to display the data, we must perform a "join" between both **Bases** and project out the replaced **Levels Descriptors** in the *multidimensional table*. In this case, the "join" must be performed through the identifier **Descriptors** of **Levels** replaced and **Levels** introduced. Formally, let $\mathcal{A}$ be the *multidimensional table*, $\mathcal{B}$ the table showing the correspondence between both **Bases** and $\mathtt{d_1}, ..., \mathtt{d_n}$ the identifier **Descriptors** of those **Dimensions** introduced. In the relational algebra, it is equivalent to:
$$\pi_{Descriptors_{\mathcal{B}(\mathtt{d_1},...,\mathtt{d_n})},Measures_{\mathcal{A}}}(\mathcal{A} \bowtie \mathcal{B})$$

- Finally, pivoting just asks to reorder the **Levels** identifiers using the SQL "order by" operator, not mappable to the relational algebra. For that reason, it is not included in table 1.

– The multidimensional **Union** (**Difference**, **Intersection**) unites two **Cubes** defined over the same multidimensional space. In terms of the relational algebra, it is equivalent to "Union" two *multidimensional tables*.

3.2 The Multidimensional Algebras

Next, we present a comparative among our reference algebra and the other multidimensional algebras presented in the literature. To the best of our knowledge, it is the first comparative about multidimensional algebras carried out. In [VS99], a survey describing the multidimensional algebras in the literature is presented. However, unlike us, it does not compare them. Results presented along this section are summarized in table 3. There, rows, representing an algebraic operator, are grouped according to which algebra they belong to (also ordered chronologically), whereas columns represent the multidimensional algebraic operators in our framework. Notice **Roll-up** and **Drill-down** are considered together since one is the inverse of the other. Moreover, as discussed in section 2.2, we consider all together the **Union**, **Intersection** and **Difference** operators.

The notation used is the following; a ✓ cell means that those operations represent the same conceptual operator; a $\sim$ stands for operations with similar purpose but different proceeding making them slightly different; a $\checkmark_p$ means that the operation partially performs the same data manipulation than the $\mathbf{YAM}^2$ operator despite the last also embraces other functionalities, and a $\checkmark_+$ means that this operation is equal to combine the marked operators of our reference algebra, meaning it is not an atomic operator. Analogously, there are some $\mathbf{YAM}^2$ operators that can be mapped to another algebra combining more than one of its operators. This case is showed in the table with a $\mathcal{D}$ (from *derived*). Keep in mind this last mark must be read vertically unlike the rest of marks. Finally, notice we have only considered those operations manipulating data. Consequently, those aimed to manipulate the data structure are not included.

[LW96] introduces a multidimensional algebra as well as its translation to SQL. To do so, it previously extends the relational algebra with grouping and aggregation operators, and later, it presents the multidimensional operators translation to the grouping algebra defined. Prior to present its operators, we must notice it was one of the first models presented, and its main aim is to construct multidimensional **Cubes** from local operational databases. In fact, they provide the "Construct" operator to generate **Cubes** from relations. More precisely, it defines five multidimensional operators representing mappings between either **Cubes** or *relations* and **Cubes**.

The "Add dimension" operator adds a new **Dimension** to the current **Cube**, like in the **changeBase** operator; the "Transfer" operator rearranges data in the multidimensional space similar to a **changeBase**. This operation *transfers* a **Dimension** *attribute* (a **Descriptor**) from one **Dimension** to another

Algebra	Operator	Selection	Projection	Roll-up Drill-down	changeBase	Drill-across	Union Difference Intersection	Remarks
[LW96]	"Add Dimension"				$\checkmark_p$			
	"Transfer"				$\sim$			
	"Cube Aggr."			✓				
	"Rc-join"	✓						
	"Union"						✓	
[AGS97]	"Push"					$\checkmark_p$		Semantic Rels.
	"Pull"		$\mathcal{D}$		$\checkmark_p$			Semantic Rels.
	"Destroy Dimension"		$\mathcal{D}$		$\checkmark_p$			
	"Restriction"	✓						
	"Join"					✓		
	"Merge"			✓				
[ML97]	"Selection"	✓						
	"Projection"		✓					
	"Cartesian Product"					$\sim$		
	"Union/Diff./Inters."						✓	
	"Fold/Unfold"				$\checkmark_p$			
	"Classification"			$\mathcal{D}$				
	"Summarization"			$\mathcal{D}$				
[TD97]	"Restriction"	✓						
	"Metric Projection"		✓					
	"Aggregation"			✓				
	"Cartesian Product"					$\sim$		
	"Join"					✓		
	"Union/Diff."						✓	
	"Extract"				$\checkmark_p$			Semantic Rels.
	"Force"					$\checkmark_p$		Semantic Rels.
[Leh98]	"Slicing"	✓						
	"Roll-up/Drill-down"			✓				
	"Split/Merge"			$\sim$				
	"Implicit/Explicit Aggr."			$\checkmark_p$				
	"Cell Operators"							Derived Measures
[CT98b]	"Cartesian Product"					$\sim$		
	"Natural Join"					✓		
	"Roll-up"			$\mathcal{D}$				
	"Aggregation"			$\mathcal{D}$				
	"Level Description"				$\checkmark_p$			Semantic Rels.
	"Scalar Function App."							Derived Measures
	"Selection"	✓						
	"Simple Projection"		✓		$\checkmark_p$			
	"Abstraction"		$\checkmark_+$		$\checkmark_{p+}$			
[HS98]	"Restrict"	✓						
	"Destroy"				$\checkmark_p$			
	"join"					✓		
	"Join"			$\checkmark_+$		$\checkmark_+$		
	"Aggr"			✓				
[Ped00]	"Selection"	✓						
	"Projection"		✓					
	"Union/Diff."						✓	
	"Identity-based Join"					$\sim$		
	"Aggregate Formation"			$\checkmark_p$				
	"Value-based Join"					✓		
	"Duplicate Removal"							**cell** definition
	"SQL-like Aggr."				$\checkmark_p$			
	"Star-join"	$\checkmark_+$		$\checkmark_+$				
	"Roll-up/Drill-down"			✓				
[Vas00]	"Navigate"			✓				
	"Selection"	✓						
	"Split Measure"		✓					
[FK04]	"Derived Measures"							Derived Measures
	"Join"					$\checkmark_p$		
	"Slice/Multislice"	✓						
	"Union/Diff./Inters."						✓	
[YP04]	"Selection Cube"	✓						
	"Decoration"				$\checkmark_p$			
	"Fed. Gen. Projection"		$\checkmark_+$	$\checkmark_+$	$\checkmark_+$			

Table 3. Summary of the comparative between **YAM**^{2} and the rest of multidimensional algebras presented in the literature.

via a "Cartesian Product". Since multidimensional concepts are directly derived from non-multidimensional relations, concepts like **Dimensions** could be rather vaguely defined, justifying the transfer operator; the "Cube Aggregation" operator performs grouping and aggregation over data, being equivalent to a **Roll-up** and finally, the "Rc-join" operator, that allows us to join a *relational* relation with a **Dimension** of the **Cube** projecting (selecting) the values in the **Dimension** also present in the relation. It is a low level operator tightly related to the multidimensional model presented, and it is introduced to relate non-multidimensional relations with relations modeling multidimensionality (i.e. **Cubes**). In our framework, it is equivalent to perform a **Selection** over a certain **Dimension**.

[AGS97] presents an algebra composed by six operators rather relevant since they inspired many of the following algebras as we will see. First, "Push" and "Pull" transform a **Measure** into a **Dimension** and viceversa, since in their model **Measures** and **Dimensions** are handled uniformly. In our framework they would be equivalent to define semantic relationships between the proper **Dimensions** and **Cells** and then, **Drill-across** and **changeBase** respectively. The "Destroy Dimension" operator drops a **Cube Dimension**, like in the **changeBase** operator, whereas the "Restriction" operator is equivalent to a **Selection**, "Merge" to a **Roll-up** and "Join" to an unrestricted **Drill-across**. Consequently, the latter can even be performed without common **Dimensions** between the **Cubes**, performing a "Cartesian Product" and embracing a massive double-counting. Notice that defining the "Cartesian Product" in a general sense does not make any multidimensional sense if it is not restricted, since it does not preserve disjointness when aggregating data. Finally, we can perform a **Projection** by means of "Pull"ing the **Measure** into a **Dimension** and performing a "Destroy Dimension" operation over it.

[ML97] presents an algebra based on the classical algebraic operations. Therefore, it includes "Selection", "Projection", "Union" / "Intersection" / "Difference" and the "Cartesian Product". All of them, except for the latter, being equivalent to their analogous operator in our reference algebra. The "Cartesian Product", like in the previous algebra, is defined as a binary relationship between two **Cubes** and therefore, mappable to an unrestricted **Drill-across**. "Fold" and "Unfold" operators add or remove a **Dimension** to the multidimensional space respectively, like in a **changeBase** and finally, it presents a **Roll-up** as a "Summarization of Tables" and a "Classification of Tables", where "Summarization" summarizes data according to an aggregation function and "Classification" maps results into groups (close to the GROUP BY clause modus operandi).

[TD97] and [TD01] present an algebra with eight operators based on the algebra presented in [AGS97]. Therefore, the "Restriction" operator is equivalent to a **Selection**; the "Metric Projection" to a **Projection**; the "Aggregation" to a **Roll-up** and the "Union" / "Difference" operators to those with the same name in our reference algebra. Moreover, like in [AGS97], **Measures** can be con-

verted to **Dimensions** and viceversa (i.e. they are handled uniformly). Hence, the "Force" and "Extract" operators are equivalent to the "Push" and "Pull" operators introduced above. Finally, the "Cubic Product" is equivalent to the "Join" operator in [AGS97]. Since a general "Cartesian Product" do not make multidimensional sense, they also remark the specific case of a "Cubic Product" over two **Cubes** with common **Dimensions** (preserving disjointness if they are joined through their **Dimensions** in common). They call "Join" to this specific "Cubic Product".

[Leh98] presents an algebra composed by five operators. "Slicing" restricts the multidimensional model in the same sense than **Selection**; "Roll-up" and "Drill-down" and the "Split" and "Merge" operators are equivalent to **Roll-up** and **Drill-down**. Despite they represent the same conceptual operators, its model data structure, that differentiates two analysis phases of data, justifies them. "Roll-up" and "Drill-down" find and interesting context in the first phase whereas "Split" and "Merge" are needed to modify the data granularity *dynamically* along the "dimensional attributes" (non-identifiers **Descriptors**) defined in the "classification hierarchies" nodes of the data structure. Moreover, like in other algebras, they differentiate "Roll-up" from "Aggregation" of data. Because of that, they also present two other operations aimed to aggregate and group data, the "Implicit" and the "Explicit" aggregation. According to this, to "Roll-up" means to perform an "Implicit aggregation" according to an aggregate function defined over the multidimensional object. Finally, the "Cell-oriented operator" derive new data preserving the same multidimensional space by means of "unary operators" (-, *abs* and *sign*) or "binary operators" (*, +, -, /, *min* and *max*). "Binary operators" ask for two multidimensional objects aligned (that is, with exactly the same multidimensional space). In our framework it is obtained defining **Derived Measures** when designing the multidimensional schema, and therefore, in design time.

[CT97], [CT98a] and [CT98b] present an algebra with nine operators. "Selection", "Cartesian Product" and "Natural Join" are equivalent to those introduced along this section. Similar to [Vas00], [Ped00] and [ML97], **Roll-up** is equivalent to "Roll-up" and "Aggregation". "Roll-up" is the conceptual change of **Levels** through an aggregation relation whereas "Aggregation" aggregates and groups data according to the **Levels** and aggregation functions depicted in the "Roll-up". A "Level description" is equivalent to an specific **changeBase**. It changes a **Level** by another one related through a one-to-one relation to it. In our framework we should define a semantic relationship among the **Levels** involved and perform a **changeBase**. "Simple projection" projects out the selected **Measures** and reduce the multidimensional space dropping **Dimensions**. Moreover, it can only drop **Measures** or **Dimensions** or combine both. To drop **Measures** is equivalent to a **Projection** and to drop **Dimensions** to a **changeBase**. Finally, "Abstraction" is equivalent to the "Pull" operator in [AGS97].

[HS98] presents a Description Logics based algebra developed from those presented in [AGS97]. Therefore, it also introduces the "Restrict" operator; the "Destroy" one equivalent to the "Destroy Dimension" and the "Aggr" operator equivalent to a "Merge". Furthermore, the "join" and "Join" operators can be considered an extension of the "Join" operator in [AGS97]. Both operators restrict it to make multidimensional sense and consequently, being equivalent to a **Drill-across**, despite the second one also allows to group and aggregate data before showing data. Consequently, it is equivalent to a **Drill-across** and **Roll-up**.

[Ped00] presents an algebra where "Selection", "Projection", "Union" / "Difference" and **Roll-up** and **Drill-down** are equivalent to those with the same name presented in our framework, whereas the "Value-based join" is equivalent to a **Drill-across** and the "Identity-based join" to a "Cartesian product". As already presented in other algebras, it also differentiates the "Aggregate operation" from the "Roll-up" one and two different operators are introduced. It also introduces the "Duplicate Removal" operator to remove **cells** characterized by the same combination of dimensional values. In our framework it can never happen because of the **Base** definition introduced.

Finally, it presents a set of non-atomic operators; the "star-join" operator combines a **Selection** over the **Dimensions** with a **Roll-up** over a certain **Dimensions** by the same aggregation function, and the "SQL-like aggregation" applies the "Aggregate operation" to a certain **Dimensions** and projects out the rest (that is, performs a **changeBase**).

[Vas00] presents an algebra with three operators focusing on the most common multidimensional operators. "Navigation" allows us to **Roll-up**, and according to [Vas98] it is performed by means of "Level-Climbing" -reducing the granularity of data-, "Packing" -grouping data- and "Function Application" -aggregating by means of an aggregation function-. Finally, "Split a Measure" is equivalent to a **Projection** and a "Selection" to our **Selection**.

[YP04] presents an algebra over an XML and OLAP federation where "Selection Cube" is equivalent to **Selection**; the "Decoration" operator adds new **Dimension**s to the **Cube** and therefore, being mappable to a **changeBase** and the "Federation Generalized Projection" **Roll-ups** the **Cube** and removes unspecified **Dimensions** (**changeBase**) and **Measures** (**Projection**). Notice despite the **Roll-up** is mandatory in this operator, we can combine it with a **Projection** or/and a **changeBase**.

An algebra with four operations is presented in [FK04]. "Slice" and "Multislice" select a single or a range of values like a **Selection**; "Union" / "Intersection" / "Difference" are equivalent to the same operators in our reference algebra; "Join" rather close to **Drill-across** but in a more restrictive way forcing both **Cube**s to share the same multidimensional space and "Derived Measures" to derive new measures from already existent. In our framework, as already said,

it should be performed in the schema design phase. Finally, notice they do not include **Roll-up** in their set of operators. It is because it is considered in the data structure of the model.

Some of these approaches have also presented an equivalent calculus besides the algebra introduced above (like [ML97] and [CT98b]). [GMR98] presents a query language to define the expected workload for the Data Warehouse. We have not included it in table 3 since it can not be compared smoothly to algebraic operators one per one. Anyway, analyzing it, we can deduce many of our reference algebraic operators are also supported by their model like **Selection**, **Projection**, **Roll-up**, **Union** and even a partial **Drill-across** as they allow to overlap fact schemes.

3.3 Discussion

As seen in section 2, to carry out our work we need a multidimensional framework in which to compare the different multidimensional algebras available nowadays. We have chosen $\mathbf{YAM}^2$ algebra as our reference framework since, as presented in section 3.2, it embraces all the data operators presented up to now in the literature, allowing us to carry out the comparison among multidimensional algebras, as well as the comparison with the relational algebra, without loss of generality.

Along this section, we would like to underline the necessity to work in terms of a multidimensional algebra. As showed in section 3.1, the multidimensional data manipulation should be performed by a restricted subset of the relational algebraic operators; that is, an specific simplification. Therefore, we can not use the whole relational algebra expression power and it must be restricted and conditioned in order to be adapted to multidimensionality. Otherwise, the results of the operations performed either would not form a **Cube** (since they are not closed with regard to the multidimensional model) or would introduce aggregation problems (see section 6 for further details). For instance, we can not talk about "cartesian product" in the multidimensional model, and we must be restricted to "joins" to avoid double-counting instances (i.e. to preserve disjointness). In other words, the multidimensional algebra represents the relational algebra subset applicable to multidimensionality. Furthermore, a multidimensional algebra would allow us to develop easier and amicable front-ends as demanded in OLAP tools, since it would provide us with a set of operators to apply over data. For instance, we could create a front-end assigning to each operator just a button; something rather complicate to develop with declarative languages like SQL or MDX.

Moreover, as seen in section 3.2 by means of a comparison between our framework and the rest of multidimensional algebras introduced, we could not use any of those multidimensional algebras as the standard framework if we would like to embrace all data operators presented in the literature. However, in that comparison we have been able to identify some significant general trends. Firstly, **Selection**, **Roll-up** and **Drill-down** operators are considered in all the algebras. It is

quite reasonable since **Roll-up** is the main operator of multidimensionality and **Selection** is a basic one, allowing us to select a subset of multidimensional points of interest out of the whole n-dimensional space. **Projection**, **Drill-across** and **Union** are included in most of the algebras presented. In fact, along the time, just two of the first algebras presented did not include **Projection** and **Drill-across**, but since then, the rest of algebras considered them somehow. About **Union**, it depends on the transformations that the model allows us to perform over data and indeed, it is a personal decision to make. However, we do believe that to unite (intersect, difference) two **Cubes** is a kind of navigation desired and easily extensible to all the algebras presented.

Finally, **changeBase** is also considered in most of the algebras. Specifically, they agree on the necessity of modifying the n-multidimensional space adding / removing **Dimensions**, and they include it as a first class citizen operator. However, unlike **YAM**^{2}, they do not present any alternative way to rearrange the multidimensional space. Our framework proposed allows two additional alternatives: to change the multidimensional space **Base** (either replacing a **Dimension** with another one or changing the whole **Base** by an alternative one), and "pivoting", as presented in [FBSV00]. In general, we can always rearrange the multidimensional space in any way, if we preserve the functional dependencies of the **cells** with regard to the **Levels** conforming the **Cube Base**; that is, if the replaced **Dimension**(s) and the new one(s) are related through a one-to-one relationship. Anyway, the **changeBase** operator subsumes the "Add" / "Remove Dimension" operator considered in the literature and raises up new desirable alternatives to handle data.

Consequently, the comparative presented reveals many implicit agreements among all the multidimensional algebras and in fact, there are many points in common about how multidimensional data should be handled. Actually, like in the conceptual multidimensional modeling issue, we strongly believe it could be feasible to agree on a reference multidimensional algebra; crucial for the evolution of the area. Ideally, the standard multidimensional algebra would need to be subsumed by the relational algebra and, at the same time, subsuming all the multidimensional algebras. In that case, it would give support to all the multidimensional data operators presented in the literature.

4 Correspondence Between the Multidimensional Algebra and SQL

This section presents in detail how the **YAM**^{2} multidimensional algebra should be translated to SQL. With this aim, we first present the template query (also known as cube-query), using the standard SQL'92, to retrieve a **Cell** of data from the RDBMS:

```
SELECT l1.ID, ..., ln.ID, [ F( ]c.Measure1[ ) ], ...
FROM Cell c, Level1 l1, ..., Leveln ln
WHERE c.key1=l1.ID AND ... AND c.keyn=ln.ID [ AND li.attr Op. K ]
[ GROUP BY l1.ID, ..., ln.ID ]
[ ORDER BY l1.ID, ..., ln.ID ]
```

The FROM clause contains the "Cell table" and the "Level tables". These tables are properly linked in the WHERE clause as well as logic clauses restricting an specific **Level** attribute (i.e. a **Descriptor**) to a constant $\mathcal{K}$ by means of a comparison operator (i.e. equality, inequality, major, minor, etc.). The GROUP BY clause shows the identifiers of the **Level**s at which we want to aggregate data. Those columns in the grouping must also be in the SELECT clause in order to identify the values in the result. Finally, the ORDER BY clause is intended to sort the output of the query by these identifiers. Notice the GROUP BY clause forces to aggregate **Measures** by means of aggregation functions, if present. Otherwise, the GROUP BY clause is not necessary and it would be redundant, since no data aggregation is performed.

This template allows us to retrieve all **cell**s of a **Cell** which conform a **Cube** that can be manipulated through the multidimensional operators, which will modify appropriately the initial cube-query. Hence, we talk about *atomic cube-query* when it just retrieves a **Cube** of data not yet manipulated by multidimensional operations.

Next, we present how each multidimensional operator (presented previously in section 2.2) modifies the atomic cube-query, summarized in table 4.

Clause	**ChangeBase**	**Drill-across**	**Selection**	**Roll-up**	**Projection**	**Union**	
SELECT	Replace (LevelID)	Add (Measure)		Replace (LevelID)	Remove (Measure)		
FROM	Add (Levels)	Add (Cell)				Union (Cells and Levels)	
WHERE	Add (links)	Add (links)	AND (conditions)			Union (links)	OR (conditions)
GROUP BY	Replace (LevelID)			Replace (LevelID)			
ORDER BY	Replace (LevelID)			Replace (LevelID)			

Table 4. SQL query sentence modifications according to each multidimensional operation

- **Selection**: In SQL, it means to *and* the corresponding clause to the WHERE clause. For instance, as presented in table 5.a, we can select those `Weekly Stocks` referring to `cookies` in the `Product` **Dimension**.
- **Roll-up**: In SQL, it changes the identifier in the GROUP BY clause by that of the parent **Level**. Thus, SELECT and ORDER BY clauses must be modified accordingly, so that the **Descriptor**s coincide in all three. Measures in the SELECT clause must also be summarized using an aggregation function. To roll up to **Level All**, all **Descriptor**s of a **Dimension** are removed from the GROUP BY, and "All" is placed in the corresponding place in SELECT clause. Going on, we can **Roll-up** from `City` to **Level** `All` along `Place` **Dimension** (Table 5.b).
- **ChangeBase**: In SQL it can be performed in two different ways. Firstly, it means to reorder **Level** identifiers in ORDER BY and SELECT clauses when "pivoting". Secondly, in case of changing to another **Base**, it means

to add the new **Level** tables to the FROM and the corresponding links to the WHERE clause. Moreover, identifiers in the SELECT, ORDER BY and GROUP BY clauses must be replaced consonantly. Following with the same example, we can change from (`Product` × `Week` × `All`) to (`Product` × `Week`) **Base** and therefore, preserving the functional dependence and not losing **cell**s (Table 5.c).

- **Drill-across**: In SQL, it means to add a new **Cell** table to the FROM, its **Measure**s to the SELECT, and the corresponding links to the WHERE clause. In general, if we are not using any *semantic relationship*, a new **Cell** table can always be added to the FROM clause if the attributes composing the identifier of the desired **Cell** point to the already used **Level** tables. For instance, in the same example, we could **Drill-down** to `Daily Stock` and directly **Drill-across** to `Daily Profit` (Table 5.d).
- **Projection**: In SQL it removes **Measures** from the SELECT clause. Following our example, we can remove the `Stock` **Measure** (Table 5.e).
- **Union**: In SQL, we unite both FROM clauses, WHERE links, and finally *or* conditions of WHERE clauses. Hence, we can unite our example query to one identical but querying for `chocolate` instead of `cookies` (Table 5.f). As previously stated in section 2.2, these considerations can be easily extended to **Difference** and **Intersection**.

```
SELECT p.ID, w.ID, c.ID, AVG(s.Stock)
FROM weeklyStock s, Product p, Week w, City c
WHERE s.key1 = p.ID AND s.key2 = w.ID
AND s.key3 = c.ID AND p.name = 'cookies'
GROUP BY p.ID, w.ID, c.ID
ORDER BY p.ID, w.ID, c.ID
```

a) Selection

```
SELECT p.ID, w.ID, ''All'', SUM(s.Stock)
FROM weeklyStock s, Product p, Week w
WHERE s.key1 = p.ID AND s.key2 = w.ID
AND p.name = 'cookies'
GROUP BY p.ID, w.ID
ORDER BY p.ID, w.ID
```

b) Roll-up

```
SELECT p.ID, w.ID, SUM(s.Stock)
FROM weeklyStock s, Product p, Week w
WHERE s.key1 = p.ID AND s.key2 = w.ID
AND p.name = 'cookies'
GROUP BY p.ID, w.ID
ORDER BY p.ID, w.ID
```

c) ChangeBase

```
SELECT p.ID, d.ID, AVG(s.Stock), SUM(m.profit)
FROM dailyStock s, dailyProfit m, Product p, Day d
WHERE s.key1 = p.ID AND s.key2 = d.ID
AND m.key1 = p.ID AND m.key2 = d.ID
AND p.name = 'cookies'
GROUP BY p.ID, d.ID
ORDER BY p.ID, d.ID
```

d) Drill-across

```
SELECT p.ID, d.ID, SUM(m.profit)
FROM dailyProfit m, Product p, Day d
WHERE m.key1 = p.ID AND m.key2 = d.ID
AND p.name = 'cookies'
GROUP BY p.ID, d.ID
ORDER BY p.ID, d.ID
```

e) Projection

```
SELECT p.ID, d.ID, SUM(m.profit)
FROM dailyProfit m, Product p, Day d
WHERE m.key1 = p.ID AND m.key2 = d.ID AND
(p.name = 'cookies' OR p.name = 'chocolate')
GROUP BY p.ID, d.ID
ORDER BY p.ID, d.ID
```

f) Union

Table 5. Example of $\mathbf{YAM}^2$ algebra translation to SQL

5 From SQL To Multidimensionality

As presented in section 3, the multidimensional algebra conceptually maps to an specific subset of the relational one. Our aim in this section is to clearly define this mapping at a logical level, going one step beyond and thoroughly analyzing the multidimensional algebra expressiveness with regard to SQL (a short version of this work can be found in [RA06]). Hence, we would be able

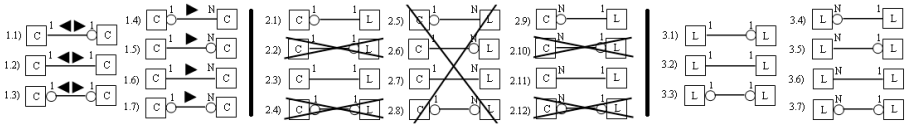

Fig. 3. Cell - Cell, Cell - Level and Level - Level relationships

to ask the following questions: Which subset of the relational algebra can be expressed in terms of the multidimensional one? Given a correct SQL query, does it make multidimensional sense?

Consequently, we determine if a correct SQL query is a *valid cube-query*. As we will see, an at first-sight syntactically correct cube-query following the multidimensional query pattern presented in section 4, may not make multidimensional sense. In fact, there are many other implicit restrictions to be guaranteed, and this pattern only guarantees the syntactic correctness of the query, not assuring its multidimensional validness. With this aim, we present those characteristics that an SQL query must enforce to make sense as a sequence of multidimensional operations. Hence, from here on, we consider a valid cube-query to be both semantically and syntactically correct in the multidimensional model.

To validate an SQL query as a cube-query we will need to find if it fits a valid multidimensional schema. Therefore, given an SQL query, our main objective will be to generate the set of multidimensional schemas validating that query. If the set obtained is empty then, it is not a valid cube-query. Otherwise, it is, and we can always find a sequence of multidimensional operations retrieving the same data than our SQL query.

Notice this analysis could have also been performed over the relational algebra instead of SQL since it is well-known how to extract the syntactic tree from a valid SQL query. However, for the sake of comprehension, we believe it is easier for the reader to reason in terms of SQL since it directly emulates the translation process a ROLAP tool would perform.

5.1 Valid Multidimensional Conceptual Relationships

Our first aim is to analyze which kind of relationships between multidimensional concepts can be found in the multidimensional schema. These relationships will be those used later by the multidimensional operators to manipulate data. Here, we first analyze them from a conceptual point of view whereas next section studies how to find and identify these relationships in a logical schema (in our case, a relational schema).

In a multidimensional schema we semantically relate **Cells** and **Levels** to analyze data contained in the firsts (**Measures**) with regard to data contained in the second ones (**Descriptors**). Therefore, we can find three kind of relationships: **Level - Level**, **Cell - Cell** and **Cell - Level** relationships.

Figure 3 summarizes all the possible relationships we can find between **Cells** and **Levels**, and an specific relationship is crossed out if it does not make sense. We say a relationship does not make sense if it violates any of the integrity constraints presented in section 2.3. Notice we have not taken into account the semantics of the relationship (for instance, an "aggregation", "association", "generalization", "derivation" or "flow" relationships as presented in [ASS06]) but it does consider the relationship multiplicity between both concepts (one-to-one, one-to-many or many-to-one) and if the relationship endings allow zeros. Keep in mind we will need to look for these relationships in a relational schema and therefore, relationships semantics will be lost but multiplicity will not. Consequently, we do not mind semantics of the relationship but if it is a valid relationship, and in this case, multiplicity is a key feature to spot the correctness of a relationship as discussed below. Moreover, notice we do not even draw many-to-many relationships since they always cause summarizability problems in the multidimensional model.

CELL - CELL relationships: Here we consider those relationships between **Cells** depicted in figure 3.1. We have conceptually divided them into two different columns based on the relationship multiplicity.

First column shows possible one-to-one relationships. It means, one instance is related with, at most, one instance from the other **Cell**, never raising summarizability problems. These cases are only possible relating both **Cell**s through their **Bases**, and therefore, if they are sharing exactly the same multidimensional space. For instance, these kind of relationships are typically used by a **Drill-across**.

Second column summarizes, according to the navigation order between **Cells**, the one-to-many and the many-to-one relationships. In these cases, since one instance of a **Cell** matches many instances from the other one, these relationships may cause summarizability problems when aggregating **Measures** through them. However, we can avoid these problems and allow these relationships in the schema, ensuring **Measures** of the *origin* **Cell** are not selected by the user when navigating through them. That is, if they have been projected out. With this constraint, to navigate trough one-to-many relationships must be forbidden, since it is meaningless to **Drill-across** to another **Cell** to just get rid of its **Measures**. Consequently, unlike one-to-one relationships, these relationships set up an strict navigation order (i.e. from many to one) between both **Cells**. Notice we talk about *navigation order* between **Cells** since they contain multidimensional data and relating them, we are implicitly setting which is the *initial* and the *destination* **Cell** of the navigation (i.e. of the **Drill-across**).

Finally, when navigating to any kind of relationship end allowing zeros (see cases 1.1, 1.3, 1.5 and 1.7), we need to bear in mind that we may lose instances from the *origin* **Cell**. In general, and similarly in the rest of upcoming cases, ROLAP tools should preserve the multidimensional space when navigating to a side allowing zeros.

CELL - LEVEL relationships: These relationships, presented in figure 3.2, are the most common multidimensional relationships in the schema. They are intended to show the **Cell** data depending on its analysis **Level**s perspective and, unlike previous cases, these relationships semantics do not set up a navigation order.

First column shows those one-to-one relationships. One **Cell** instance is related to at most one **Level** instance (something necessary to ensure that the analysis **Level**s do define the multidimensional space), and one **Level** instance is just related to one **Cell** instance. Although nothing prevents us from having a one-to-one **Level** - **Cell** relationship, these are rather rare. Anyway, they are possible, except for 2.2 and 2.4 cases since every **Cell** instance has to be related, at least, to a **Level** instance.

Second column is completely forbidden. Relating one **Cell** instance to many **Level** instances would imply double-counting instances when aggregating this **Cell Measures**. At most, a **Cell** instance must match one **Level** instance to ensure it is uniquely identified in the multidimensional space.

Finally, last column shows many-to-one relationships. These are the most common and typical relationships between **Cell**s and **Level**s. One cell is related to just one **Level** instance and one **Level** instance may be related to many different **Cell** instances. Similar to cases 2.2 and 2.4, cases 2.10 and 2.12 cannot be found in a multidimensional schema.

LEVEL - LEVEL relationships: Figure 3.3 shows all possible **Level** - **Level** relationships. **Level**s analyze data (i.e. **Cells Measures**) from a conceptual perspective of view. **Cell** data related to them is aggregated accordingly, and as presented below, they may cause potential aggregation problems depending on the **Cells** placement.

In the first column we find the one-to-one relationships. Each **Level** instance matches with at most one instance of the other **Level**, avoiding summarizability problems. Therefore, they are always valid regardless of the **Cells** placement. As previously discussed, cases 3.1 and 3.3 require special attention to avoid losing **cells**. These relationships are typically used by a **ChangeBase**.

Second column shows the one-to-many and many-to-one relationships. These relationships may cause aggregation problems depending on the **Cells** placement. However, at this time, we are only analyzing the relationship between both **Levels**, and a priori, all of them should be allowed. In fact, many-to-one relationships are typically used by **Roll-ups** whereas one-to-many ones are commonly used by **Drill-downs**. Furthermore, notice some of the many-to-one relationships are surely pointing out some bad conceptual design decisions. Cases 3.5 and 3.7 represent those cases where some **Level** instances may not be aggregated along the other one, raising up incomplete aggregations. In this case, best solution is to include in the destination **Level** an “others” instance embracing them, avoiding to deal with zeros. Finally, again, notice cases 3.5 and 3.7 need special attention to preserve the multidimensional space.

5.2 Multidimensional Relationships at a Relational Level

Once we have determined which conceptual relationships among multidimensional concepts are allowed, we need to analyze how they can be modeled at a logical level; that is, in our case, in the relational model.

As presented in section 2.1, multidimensionality would be modeled by means of **Cell** and **Level** tables, where **Level** tables may be partially or totally denormalized according to the implementation approach followed (i.e. an snowflake, star or hybrid schema). Therefore, we need to look for those patterns in the organizational relational schemas in order to point multidimensional concepts out. Moreover, when navigating through data by means of the relational algebra, conceptual relationships presented will give rise to "joins" between relational tables. Consequently, in this section, we present those features any relational join must satisfy to model a valid relationship among multidimensional concepts. That is, if it models one of those valid relationships presented in figure 3.

First feature is about semantics, essential in multidimensionality. When joining two relation attributes, we should guarantee they have been defined over compatible "semantic domains" and therefore, they are semantically overlapped. For instance, it is meaningless to join `city names` with `providers names` despite being defined over the same data type. ORDBMS (*Object Relational Database Management Systems*) provide us with semantic domains, where we can guarantee the semantic correctness of a join, but the relational model does not enrich data with semantics and therefore, we are restricted to syntactic domains. However, relational systems allow us to guarantee the semantic compatibility between two attributes defining a FK (*Foreign Key*) with regard to a CK (*Candidate Key*). In our approach, we assume those joins whose semantics can not be automatically validated are also correct; since the SQL query to analyze expresses users requirements.

CK_o	CK_d	FK_o	FK_d	NN_o	NN_d	Relationship	Multiplicity
×	×	×	×	?	?	*Attr.* → *Attr.*	$N - N$
✓	×	×	✓	✓	✓	$CK \rightarrow FK + NN$	1 -o N
✓	×	×	?	✓	?	$CK \rightarrow$ *Attr.*	1 o-o N
×	✓	✓	×	✓	✓	$FK + NN \rightarrow CK$	N o- 1
×	✓	?	×	?	✓	*Attr.* $\rightarrow CK$	N o-o 1
✓	✓	✓	✓	✓	✓	$CK + FK \rightarrow FK + CK$	$1 - 1$
✓	✓	✓	×	✓	✓	$CK + FK \rightarrow CK$	1 o- 1
✓	✓	×	✓	✓	✓	$CK \rightarrow CK + FK$	1 -o 1
✓	✓	×	×	✓	✓	$CK \rightarrow CK$	1 o-o 1

Table 6. Relationship multiplicities in the relational model.

Next, to find out which conceptual relationships may appear in the relational model we first focus on their multiplicity. In the relational model, multiplicity

Multiplicity	L - L	C - C	L - C	C - L
1 - 1	✓	✓	✓	✓
1 o- 1	✓	✓	×	✓
N - 1	✓	✓	×	✓
N o- 1	✓	✓	×	✓
N o-o 1	✓	✓	×	×
N -o 1	✓	✓	×	×
1 o-o 1	✓	✓	×	×

Table 7. Valid multidimensional relationships in a relational schema.

of a relationship depends on how attributes involved are defined in the schema. That is, if they play the role of a relation CK and / or if they are defined as a FK to the other attribute and / or if they allow null values. Joining to a CK guarantees to match at most one instance of the relation. Otherwise it may match many of them. Similarly, an attribute not allowing null values and being defined as FK will surely match one and just one instance. Otherwise, it may introduce zeros. Table 6 summarizes all those relationship multiplicities that we may find in the relational model with regard to the definition of the attributes involved. There, each row represents an specific relationship between tables (i.e. a kind of join).

The notation used is the following; first six columns represent all possible combinations with regard to the constraints of each join attribute: As CK, as a FK pointing to the other attribute or as a NN (*not null*) attribute, not allowing null values. If an specific cell is ✓, the attribute is constrained accordingly to that column. Otherwise, it is marked with a × mark. Notice not all the combinations are correct and some columns determine the following ones. For instance, a CK attribute can not accept null values. Moreover, a cell is marked with a ? mark if previous rows determine a certain multiplicity, meaning its value does not affect the result obtained. Finally, two last columns tell us the specific join depicted as well as its multiplicity with regard to previous columns. There, an `Attr.` represents an unconstrained attribute; that is, not defined neither CK nor FK and allowing null values.

First row represents a join between two attributes not defined as CK's. Therefore, they can not be defined as FK's to the other attribute and it does not matter if they allow not null values since they will always raise a many-to-many relationship, not allowed in multidimensionality. Following two rows identify those joins where the origin attribute is defined as a CK but not the destination one. In this case, every instance of the destination table matches, at most, one instance from the origin one. Furthermore, if the destination attribute is defined as a FK not allowing nulls we can also ensure it matches, at least, one instance from the origin table. Otherwise, it may give rise to zeros in the right-side of the relationship. Finally, notice 1 - N and 1 o- N relationships can not be enforced in the relational model just checking the schema (we do not consider neither as-

sertions nor triggers). Next rows represent an unconstrained attribute linked to a CK. That is, the same two previous relationships presented but the other way around, giving rise to the same considerations. Last four rows identify those relationships among two CK's, raising up one-to-one relationships. In these cases, depending on the FK's defined we may introduce zeros to each relationship side.

Once we know which relationships multiplicity can be found in a relational schema, we need to validate them according to the roles played by the tables involved. Table 7 summarizes those multidimensional relationships we can find in a relational schema. There, columns present all possible relations between **Cell** and **Level** tables (see section 5.1 for more details) whereas rows represent those multiplicities we may face in the relational model. An specific cell is marked with a ✓ mark if that multidimensional relationship along with the stated multiplicity makes multidimensional sense. Otherwise, it is marked by a ×, meaning it must be avoided. Therefore, notice this table merges results presented in figure 3 and table 6.

As discussed previously, we can not differentiate between a 1 - N and 1 -o N relationship or a 1 o- N and a 1 o-o N relationship just looking at a relational schema. However, as presented in table 7, they give rise to the same results (i.e. rows 3-4 and rows 5-6 are identical), not affecting our process at all. Furthermore, as presented in section 5.1, when navigating to a relationship side allowing zeros, we must enforce the user to "left-outer join" both tables in order to preserve the multidimensional space.

5.3 Analyzing the Cube-Query

In this section we validate a syntactically correct SQL query as a valid cube-query. To do so, we need to find if it fits a multidimensional schema. Therefore, starting from an input SQL query, we automatically generate the set of multidimensional schemas validating that query. If the input query is not multidimensional (i.e. it does not represent multidimensional requirements) we will not be able to propose any schema. Multidimensional schemas proposed will be inferred from those implicit restrictions the SQL query needs to guarantee to make multidimensional sense. Furthermore, in this process, the organizational database schemas will play a key role as we will see.

To start this process we first create what we call the *multidimensional graph*; that is, a graph representing the multidimensional query. Representing the query by means of a graph will help us to facilitate the validation of the query, since it concisely stores relevant information about the query.

The graph is deployed along four steps and it is composed of *nodes*, representing tables involved in the query and *edges*, relating nodes (i.e. tables) joined in the query. As stated in section 2.1, in the relational model multidimensionality is modeled through **Cell** and **Level** tables. Therefore, tables appearing in a cube-query would play either a **Cell** or a **Level** role.

Moreover, each node contains three properties needed along the validation process. The *name* property stores the table name, and will be used as its identifier along the process. The *type* property identifies that node as a **Cell** if labeled

with a $\mathcal{C}$, as a **Cell** with selected **Measures** (i.e. at least one of its **Measures** appear in the SELECT clause), if labeled with a $\mathcal{CM}$ or as a **Level** table if labeled with an $\mathcal{L}$. Finally, the *attribute list* property stores all those table attributes selected in the query. Analogously, for each edge we also need to store three properties. The *navigation* property sets up the navigation order (i.e. a directed arrow or a bidirectional one); the *valid relationships list* property setting those allowed relationships between nodes related and at last, the *join attributes* property, storing those attributes involved in the join.

Next, we present the steps to create such multidimensional graph. Each step is aimed to validate each clause in the cube-query and to extract relevant knowledge from it to be represented in the graph. Notice we do not validate it syntactically, since we assume it is a correct SQL query and consequently, interpretable by a RDBMS. Therefore, we focus on its multidimensional semantics correctness. Finally, since this process is thought to be performed automatically, we present it all together with an algorithm, described in pseudo code.

1. **For each** table **in the** FROM clause **do**
   (a) **Create** a node;
   (b) **Initialize** node properties;
2. **For each** attribute **in the** GROUP BY clause **do**
   (a) node = $get\_node(attribute)$;
   (b) **if** (!$defined\_as\_part\_of\_a\_CK(attribute)$) **then**
      i. **Set** *node* type property **as Level**;
   (c) **else if** (!degenerate dimensions allowed) **then**
      i. FK = $get\_FK(attribute)$;
      ii. $node\_dest$ = node;
      item $attributes\_FK$ = attribute;
      iii. while $chain\_of\_FKs\_follows(FK)$ and $FK\_in\_WHERE\_clause(FK)$ **do**
         A. FK = $get\_next\_chained\_FK(FK)$;
         B. $node\_dest$ = $get\_node(get\_table(FK))$;
         C. $attributes\_FK$ = $get\_attributes(FK)$;
      iv. /* We must also check #attributes selected matches #attributes at the end of the chain. */
      v. **if** (FK == **NULL** and $\#attrs(attribute) == \#attrs(attributes\_FK)$) **then**
         A. **Set** $node\_dest$ type property **as Level**;
3. **For each** attribute **in the** SELECT clause **not in the** GROUP BY clause **do**
   (a) node = $get\_node(attribute)$;
   (b) **Set** *node* type **property to** CM; //Cell with Measures selected

Fig. 4. Three first steps of the process.

Step 1, the FROM clause: As stated in figure 4, for each table in the FROM clause we create a node setting its name property to the table name, its type property to the ? mark (i.e. unknown) and its attribute list property to the empty set. Along the process, our main objective will be to label nodes according to its role played. In a certain moment, if a node has been already labeled and we need to label it with a different tag, we finish the process and point out the contradiction stated.

Step 2, the GROUP BY clause: This clause contains those attributes depicting the multidimensional space (i.e. the current **Cube Base** composed by those **Dimensions** of analysis fully functionally determining data). We consider a **Cell** table to always point out to its **Dimensions** of analysis, as

typically assumed when modeling multidimensionality (for instance, in an star schema). Therefore, if an attribute is not defined as FK or it is but we are able to follow a FK's chain defined in the schema that is also present in the WHERE clause, we can state for sure that the table where the FK's chain ends plays a **Level** role. Conversely, if it is part of a CK, we can directly state that that table is a **Level**.
Notice we allow multiattribute FK's. In that case, the whole FK must appear properly linked and those attributes reached at the end of the FK's chain must match (in number of attributes) those in the GROUP BY.

Step 3, the SELECT clause: Since a syntactically correct SQL query forces both the GROUP BY and SELECT clauses to share the same attributes, we can assure those attributes in the GROUP BY represent the **Cube Base** whereas those aggregated attributes in the SELECT clause not present in the GROUP BY point out **Measures**. Hence, those aggregated attributes in the SELECT clause not present in the GROUP BY clause surely play a **Measure** role. Therefore, we set the node type property with the $\mathcal{CM}$ label, denoting, unequivocally, it is a **Cell** with selected **Measures**.

4. **For each** comparison **in the** WHERE clause **do**
   (a) node = $get\_node(attribute)$;
   (b) **if** (!$defined\_as\_part\_of\_a\_CK(attribute)$) **then**
      i. **Set** *node* type property **as Level**;
   (c) **else if** (!degenerate dimensions allowed) **then**
      i. attribute = $get\_attribute(comparison)$;
      ii. FK = $get\_FK(attribute)$;
      iii. $node\_dest = get\_node(attribute)$;
      iv. $attributes\_FK$ = attribute;
      v. while $chain\_of\_FKs\_follows(FK)$ and $FK\_in\_WHERE\_clause(FK)$ **do**
         A. FK = $get\_next\_chained\_FK(FK)$;
         B. $node\_dest = get\_node(get\_table(FK))$;
         C. $attributes\_FK = get\_attributes(FK)$;
      vi. **if** (FK == **NULL** and $\#attributes(attribute) == \#attributes(attributes\_FK)$) **then**
         A. **Set** $node\_dest$ type property **as Level**;

Fig. 5. Fourth step of the process.

Step 4, selection comparisons in the WHERE clause: In a WHERE clause we can find two different kinds of clauses; comparisons over the table columns and links to *join* tables. In this step we focus on the first ones, whereas next step focus on the joins performed. Comparison clauses are intended to select data constraining a **Descriptor** to a concrete value or set of values, and must be constituted of one attribute comparisons. That means, we can only find equality (*column* = K), inequality (*column* <> K) major (*column* > K) and minor (*column* < K) comparisons, where K is a constant. In this case, as presented in figure 5, we can identify **Levels** following FK's as stated in the second step. However, notice this step will only detect new **Levels** (i.e. not detected in previous step) if the attribute compared is not included in the SELECT or the query does not contain a GROUP BY clause.

```
5. For each join in the WHERE clause do
   (a) /* Notice a conceptual relationship between tables may be modeled by several joins in the WHERE */
   (b) set_of_joins = look_for_related_joins(join);
   (c) multiplicity = get_multiplicity(set_of_joins);
   (d) relationships fitting = {};
   (e) For each relationship in get_allowed_relationships(multiplicity) do
          i. if (!contradiction_with_graph(relationship)) then
                A. relationships fitting = relationships fitting + {relationship};
   (f) if (!sizeof(relationshipsfitting)) then
          i. return notify_fail("Tablesrelationshipnotallowed");
   (g) Create an edge(get_join_attributes(set_of_joins));
   (h) Set edge valid relationships list property to relationships fitting;
   (i) if (unequivocal_knowledge_inferred(relationships_fitting)) then
          i. propagate knowledge;
```

Fig. 6. Fourth step of the process.

Step 5, joins in the WHERE clause: Previous steps are mainly aimed to label nodes whereas this step labels edges. Relationships between tables give us relevant information we need to exploit in order to validate those joins performed in the WHERE clause by means of table 7 (see sections 5.1 and 5.2 for further details). Hence, as presented in figure 6, for each join performed, we first infer the relationship multiplicity with regard to the definition of the join attributes in the schema (i.e. as FK's, CK's or Not Null). According to the relationship multiplicity, we look for those allowed relationships (with a ✓ mark in that row) depicted in table 7. For each allowed relationship for this multiplicity, we see if it contradicts our previous knowledge, that is if (1) it asks for labeling a node already labeled with a different tag. If it (2) does not preserve the multidimensional space (see section 5.1), however, we do not invalidate the query but inform the user to solve that problem. That is, in the relational model, to outer join when navigating to a relationship end allowing zeros. Otherwise, we add it to the edge multidimensional relationships property. After checking all of them, if the allowed relationships set is empty we can state this query does not make multidimensional sense. Otherwise, we set the edge joining attributes property to those attributes involved in the join. Furthermore, if we are considering just one possible valid relationship, or we can infer unequivocal knowledge (i.e. despite having some different alternatives, we can assure that origin/destination/both node(s) must be a **Cell** or a **Level**), we update the graph labeling the nodes accordingly. If we update one such node, we must propagate in cascade new knowledge inferred to those edges and nodes directly related to those elements updated.

Finally, we focus on some additional considerations. These considerations affect the process somehow and despite being rather unusual, they must be taken into account:

Queries without GROUP BY clause: If data retrieved (i.e. **Measures**) is not grouped by, we are not forced to aggregate them by means of aggregation functions in the SELECT clause and therefore, step two would not be able to point them out. Therefore, an unconstrained attribute in the SELECT clause could be either a **Measure** or a **Descriptor**. Furthermore, we would not be

able to follow FK chains either, since **Measures** could be also modeled, in the source operational schema, as FK's (for instance, to semantically restrict their values).
In addition, if the query contains a GROUP BY clause, notice every **Cell** detected after step two is automatically labeled with a $\mathcal{C}$ tag. Conversely, it does not happen if no GROUP BY clause is stated, and therefore, we could set up its type property either to $\mathcal{C}$ or to $\mathcal{CM}$. If the **Cell** attribute list property is not empty (i.e. some of its attributes are selected), we may identify any of these attributes to be a **Measure** if it is not defined as part of a CK. Otherwise, we can assure it does not select any of its **Measures**.

Degenerate Dimensions: Up to now, as discussed in 2.1 and presented in [KRTR98], we have considered a **Cell** CK points to its analysis **Levels** CK's by means of FK's. However, in a non-multidimensional relational schema this may not happen. For some reason, we could have a table attribute representing a **Dimension** not pointing to any table. For instance, dates or control numbers (like invoice number, bill of landing, etc.) are good candidates. Furthermore, it could also happen if the schema is not well-formed (i.e. the table exists but they are not linked by means of a FK). This situation, despite being rather unusual, can also appear in the multidimensional model giving rise to "degenerate dimensions" ([KRTR98]). "Degenerate dimensions" represent those **Dimensions** without **Descriptors** (maybe because their related attributes gave rise to other **Dimensions**) that are still useful for grouping data. Consequently, they are directly modeled in the **Cell** table.
If the relational schema allows "degenerate dimensions" we can not assume anymore FK's end up in **Level** tables. Consequently, steps two and three of the process are directly affected as depicted in figure 4 (step 2c) and figure 5 (step 4c) respectively.

5.4 Validating the multidimensional graph

Once the multidimensional graph has been deployed, we need to validate if it represents a correct multidimensional schema as a whole. However, notice the graph construction may have not labeled all the nodes. By means of backtracking, we first look for all those valid labeling alternatives, and any labeling giving rise to contradictions (see previous subsection) is eluded. If the process ends without being able to label all the nodes at least once, we can assure there is not any multidimensional schema validating the input query. Otherwise, this retrieves all those multidimensional graphs composed of valid edges, and each one of them needs to be validated as a whole (see figure 7).

Following, we present those steps aimed to validate the multidimensional graph:

Step 6, the graph must be connected: In general, the multidimensional graph must be *connected* to avoid the "Cartesian Product" among tables involved in the query. Furthermore, the graph must look like as the one presented in

```
6. If !connected(graph) then
   (a) return notify_fail("Graph not connected.");
7. For each subgraph of Levels in the multidimensional graph do
   (a) if (redundant_descriptors_selected(subgraph) ||
       selecting_descriptors_from_different_branches(subgraph)) then
         i. return notify_suggestion("Descriptors subset chosen must be reconsidered");
   (b) if contains_cycles(subgraph) then
         i. /* Alternative paths must be semantically equivalent and hence raising the same multiplicity. */
        ii. if contradiction_about_paths_multiplicities(subgraph) then
              A. return notify_fail("Cyclescannotbeusedtoselectdata.");
       iii. else
              A. ask user for semantical validation;
   (c) if exists_two_Levels_related_same_Cell(cycle) then
         i. return notify_fail("Non-orthogonal Analysis Levels");
   (d) For each relationship in get_1_to_N_Level_Level_relationships(subgraph) do
         i. if left_related_to_a_Cell_with_Measures(relationship) then
              A. return notify_fail("Aggregation Problems.");
8. For each Cell pair in the multidimensional graph do
   (a) For each 1_1_correspondence(Cellpair) do
         i. Create context edge between Cell pair;
   (b) For each 1_N_correspondence(Cellpair) do
         i. Create directed context edge between Cell pair;
   (c) If exists_other_correspondence(Cellpair) then
         i. return notify_fail("Invalid correspondence between Cells.");
9. if contains_cycles(Cells path) then
   (a) if contradiction_about_paths_multiplicities(Cells path) then
         i. return notify_fail("Cycles can not be used to select data.");
   (b) else
         i. ask user for semantical validation;
        ii. Create context nodes(Cells path);
10. For each element in get_1_to_N_context_edges_and_nodes(Cells path) do
   (a) If CM_at_left(element) then
         return notify_fail("Aggregation problems among Measures.");
11. If exists_two_1_to_N_alternative_branches(Cells path) then
   (a) return notify_fail("Aggregation problems among Cells.");
```

Fig. 7. Process to validate the multidimensional graph.

figure 8; that is, the graph must be composed of valid edges giving rise to a path among **Cells** and connected subgraphs of **Levels** surrounding it. There, **Cells** contain multidimensional data to be retrieved whereas subgraphs of **Levels** point the multidimensional space out (i.e. depicting the **Dimensions** of analysis). In next steps, we will formally validate these features in detail.

Step 7, validating subgraphs of Levels: **Dimensions** of analysis should be orthogonal. Despite it could be possible to find **Dimensions** determining others in the schema, it must be avoided among **Dimensions** arranging the multidimensional space in a cube-query, in order to guarantee **cells** are fully functionally determined by **Dimensions** ([Abe02]). Unfortunately, relational schemas do not capture all data semantics and we are not able to assure that **Dimensions** selected are orthogonal. Consequently, we may select any **Level Descriptor** in a subgraph of **Levels**. However, as checked in step 7a, once we have shown (i.e. selected) a **Level Descriptor**, showing others related to this by a one-to-many relationship is absolutely redundant. Furthermore, two alternative branches in the subgraph selecting **Descriptors** (see subgraph `LS3` or `LS5` in figure 8) clearly state that those **Dimension** tables conform a hierarchy to be reconsidered (in fact, we are selecting the "Cartesian Product" of both selected **Levels**).

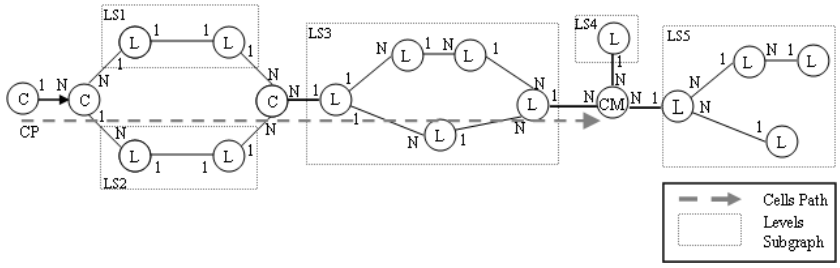

Fig. 8. A Multidimensional Graph.

If the subgraph contains a **Levels** cycle (see subgraph `LS3` in figure 8), we must assure that exist two **Levels** (conceptually the *top* and *bottom* of the hierarchy) so that every path between them must be semantically equivalent (step 7b). It would mean that such a cycle represents a valid **Dimension** hierarchy. If the subgraph contains more than one cycle, we just need to focus on the biggest one embracing the rest of cycles and validate it, since contained cycles will just represent new alternative paths to be validated. Consequently, to validate cycles we must be able to spot the *origin* and *destination* nodes of the cycle so that every instance in the origin node matches the same instances in the destination node, for every possible path between them. Otherwise, they would be selecting data retrieved equally by those paths. Conceptually, the origin and destination node represent, respectively, the top and bottom **Levels** in the hierarchy of **Levels** depicted by that cycle. Therefore, despite we can not automatically validate cycles semantically (the user must confirm the cycle correctness), both nodes (i.e. the origin and destination) must be related by the same multiplicity for every possible path between them. This constraint could be relaxed depending on the DW criteria and allow cycles not being semantically equivalent to select data. However, in any case, we must tell the user that those joins in the WHERE clause are, indeed, a selection, and it should be performed by means of comparison clauses.

Once we have validated the subgraph of **Levels** per se, we focus on validate them with regard to **Cells** (specifically, to those **Cells** with **Measures** selected). Those subgraphs are aimed to place data in the multidimensional space and therefore, we need to assure that the data placement is free of summarizability problems.

As discussed in section 5.1, one-to-one relationships between **Levels** never raise summarizability problems, but similar to **Cell** relationships, one-to-many **Level - Level** relationships may cause aggregation problems depending on the **Cells** placement with regard to them. Specifically, if a **Cell** is related to the right side of a one-to-many **Level - Level** relationship (see step 7d), every **Cell** instance would be related to at most one instance of the left-side **Level**, avoiding them to be counted more than once. Conversely, if the **Cell** is related to the left side, it will always raise problems invalidat-

ing it: if data is grouped by means of a **Descriptor** of the left-side **Level**, it would cause summarizability problems. In this case, we may only navigate through these relationships if we ensure no data (i.e. the **Measures**) of that **Cell** is selected. Oppositely, if data is grouped by a **Descriptor** of the right-side **Level**, we would be asking to decompose data into a lower level of granularity that we can not provide (i.e. when **Drill-downing** we need that data to be materialized). Finally, since each connected subgraph of **Levels** represents a single perspective of view to analyze data, two different **Levels** in a subgraph can not be related to the same **Cell** (see step 7c); otherwise they would not be orthogonal, violating the **Base** integrity constraint.

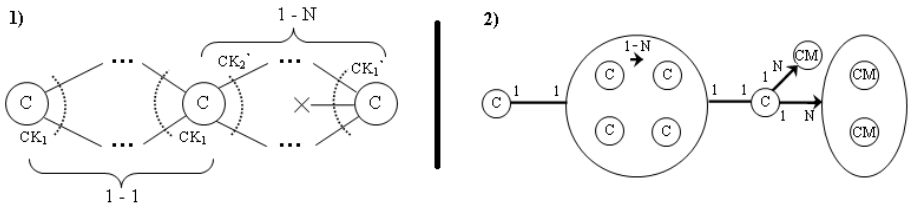

Fig. 9. Examples of **Cells** paths of context edges and nodes

Step 8, pointing out the Cells path: **Cells** determine multidimensional data and therefore, they must be related somehow (by means of direct links or / and through subgraphs of **Levels**). Otherwise, they would not retrieve a single **Cube** of data. Consequently, for every two **Cells** in the graph, we aim to validate those paths between them as a whole, inferring and validating the multiplicity raised by all those paths together, as follows: if exists a one-to-one correspondence between two **Cells**, we replace all relationships involved in that correspondence, by a one-to-one *context edge* between both **Cells** (see step 8a). Notice a context edge replaces that subgraph corresponding to the one-to-one correspondence between both **Cells**. Specifically, as depicted in figure 9.1, it means that there are a set of relationships linking, as a whole, a **Cell** CK, also linked by one-to-one paths to a whole CK of the other **Cell**. Otherwise, if both CK's are related by means of one-to-many paths or the first CK matches the second one partially, we replace involved relationships by a one-to-many directed context edge (see step 8b). Notice that these correspondences must consider **Selections** performed in the WHERE since an equality comparison fixes that **Dimension** to a unique value. For instance, a CK matching partially another CK whose unrelated attributes are fixed to a value by means of an equality, would raise a one-to-one context edge instead of a one-to-many. Any other case invalidates the graph (see step 8c) since, as presented in section 5.1, we only allow one-to-one and one-to-many relationships between **Cells**. Eventually, we will have replaced all the graph edges by context edges.

At this moment, this step has validated and represented as context edges every multidimensional conceptual relationships between **Cells** stated in the query. However, we still need to validate the whole **Cells** path constituted by these edges with regard to multidimensionality integrity constraints, along next three steps.

Step 9, validating Cells cycles: Firstly, we must validate cycles like we have previously presented when validating **Levels** cycles. That is, we must assure that every possible path in the cycle is semantically equivalent (again, the user must confirm the cycle semantical correctness). Therefore, we must be able to point out the origin and destination nodes, and every path between them must raise up the same multiplicity. Once the cycle has been validated, **Cells** involved are clustered in a *context node* as showed in figure 9.2. Finally, we label that node with the multiplicity between the origin and the destination nodes. That is, either one-to-one or one-to-many. In the latter, we also add the navigation order. Notice that, again, we are replacing that subgraph by a node depicting the whole correspondence.

Step 10, validating the Cells path: Secondly, as discussed in section 5.1, if there are any one-to-many context edge or node in the path, any **Cell** at the left-side of that edge (or node) can not select **Measures**. That is, we propagate the one-to-many **Cell - Cell** relationship constraint by transitivity. Notice this constraint must also be satisfied by those **Cells** clustered in a context node.

Step 11, validating alternative paths: Finally, notice the **Cells** path may conform a "tree", as presented in figure 9.2. In this case, as depicted in the figure, if more than one alternative branch contains a one-to-many context edge or node, we may face summarizability problems, since **Cells** at the right-side of the one-to-many relationships would be related through a many-to-many relationship.

5.5 A practical example

In this section, we present a practical example of the method presented along this paper. In this example, we consider figure 10 (where CK's are underlined and FK's dash-underlined) to depict the organizational relational schema. Therefore, given the following requirement: `"Retrieve benefits obtained with regard to supplier ABC, per month"`, it could be expressible in SQL as:

```
SELECT m.month, my.supplier, SUM(mp.profit)
FROM Month m, Monthly sales ms, Monthly supply my, Monthly profit mp, Supplier s, Prodtype pt, Product p
WHERE mp.month = ms.month AND mp.product = ms.product AND s.month = m.month AND ms.product = p.product AND my.month = m.month
AND my.supplier = s.supplier AND my.prodtype = pt.prodtype AND p.prodtype = pt.prodtype AND s.supplier = 'ABC'
GROUP BY m.month, my.supplier
ORDER BY m.month, my.supplier
```

To validate that multidimensional requirement, we will follow the algorithm presented along previous section. First, we start constructing the multidimen-

```
Prodtype(prodtype)
Supplier(supplier, name, city)
Product(product, prodtype (→prodtype.prodtype), discount)
Month(month, numdays, season)
Monthly profit(month (→month.month), product(→product.product), profit)
Monthly sales(month (→month.month), product(→product.product), sales)
Monthly supply(month(→month.month),prodtype(→prodtype.prodtype),supplier(→supplier.supplier))
```

Fig. 10. The organizational relational database schema

sional graph. In our case, we do not consider degenerate dimensions (see section 5.3):

Step 1: We first create a node for each table in the FROM clause. Initially, they are labeled as unknown (?) nodes.

Step 2: First, for each attribute in the GROUP BY clause, we try to identify the role played by those tables which they belong to.
- `m.month`: This attribute belongs to the `Month` table. Since it is not part of a FK, we can directly label that node as a **Level**.
- `my.supplier`: This attribute belonging to the `Monthly supply` table is defined as a FK pointing to the `supplier` attribute in the `Supplier` table. This equality can be also found in the WHERE clause, and therefore, we can follow the FK chain to the `Supplier` node, where the FK chain ends. Consequently, we label the `Supplier` node as a **Level**.

Step 3: For each attribute in the SELECT not in the GROUP BY (i.e. `mp.profit`), we identify the node it belongs to as a **Cell** with **Measures** selected.

Step 4: In this step, we analyze the `s.supplier = 'ABC'` comparison clause. First, we extract the attribute compared (`supplier`) and identify the table it belongs to (`Supplier`). Since it is not part of a FK, this table must be labeled as a **Level**. However, since it has been already labeled and there is no contradiction, the algorithm goes on without modifying the graph.

Step 5: For each join in the WHERE clause, we firstly infer the relationship multiplicity according to table 6. For instance, `mp.month = ms.month` joins two attributes that are part of two CK's in their respective tables. Therefore, we first look if the whole CK's are linked. In this case, this is true since `mp.product = ms.product` also appears in the WHERE clause. Consequently, we are joining two CK's, raising up a 1 o-o 1 relationship. Since this relationship asks to preserve the multidimensional space due to zeros, at this moment, we should suggest to the user to outer-join properly both tables.

Secondly, according to the multiplicity inferred, we look at table 7 looking for those allowed multidimensional relationships between both nodes. That is, C - C or L - L. However, last alternative raises a contradiction, since it asks to label the `Monthly profit` node as a **Level** when it has been already labeled as a **Cell** with **Measures**. Consequently, it is eluded. Since the set of relationships allowed is not empty, we create an edge and we label it accordingly.

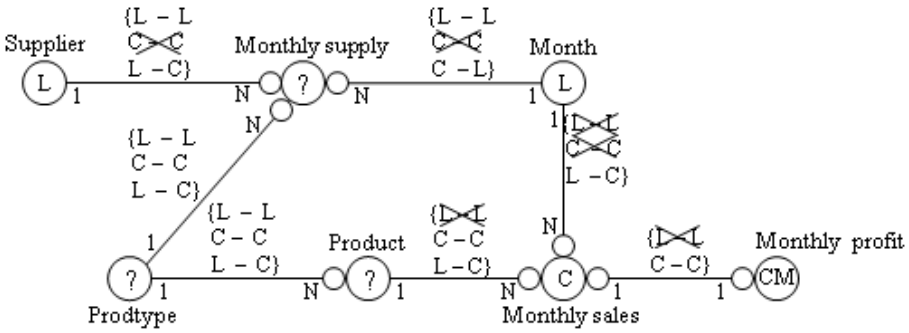

Fig. 11. The multidimensional graph deployed

Finally, we propagate current knowledge. That is, according to that edge, the **monthly sales** table must also be a **Cell**, and therefore, it is labeled as a **Cell** without selected **Measures**. After repeating this process for every join, we would obtain, at the end of this step, the graph depicted in figure 11.

At this moment, we have deployed the multidimensional graph that in next steps, we want to validate. However, since some nodes have not been labeled, we previously find out all the valid alternatives by means of a backtracking algorithm. For instance, if the `Product` node was labeled as a **Level**, according to the edge between `Product` and `Prodtype`, the latter should be also labeled as a **Level**. Moreover, the `Monthly supply` node may be labeled as a **Cell** or a **Level**. The backtracking algorithm ends retrieving all those valid labeling alternatives depicted in table 8. Notice those crossed out are eluded in this step since they raise up contradictions.

For each labeling alternative retrieved by the backtracking algorithm, we try to validate the graph. For instance, we will follow in detail the validation

Monthly supply	Prodtype	Product	Remarks
C	C	C	Illegal context edge
L	L	C	Invalid subgraph of Levels
C	L	C	Illegal context edge
L	L	L	Non-orthogonal dimensions
C	L	L	✓
C	C	L	×
L	C	C	×
L	C	L	×

Table 8. Labeling alternatives retrieved

algorithm with the first alternative, where all three unknown nodes are labeled as **Cells**:

Step 6: First, we check if the graph is connected (in this case, it is).

Step 7: In this step we validate each subgraph of **Levels** (those two depicted in figure 11). Since they do not contain cycles of **Levels** (step 7b) nor contain alternative branches (step 7a), both are correct. Next, we validate subgraphs of **Levels** with regard to **Cells**. There is neither two **Levels** in the same subgraph related to the same **Cell** (step 7c) nor **Level** - **Level** relationships (step 7d), both are correct again.

Step 8: Here, we create the context edges between **Cells**. In this case, we are not able to replace all the edges, since the `Monthly supply` and `Monthly sales` unique correspondence (thorugh the `Month` node) can not be replaced by a context edge (step 8c).

Since we have found a contradiction, we elude this labeling and try the next one. We address the reader to follow the algorithm with the rest of alternatives. Second labeling is forbidden because of step 7d, since it raises a one-to-many **Level** - **Level** (i.e. `Monthly supply` - `Month`) where the one side is related to a **Cell** with selected **Measures** (i.e. `Monthly profit`). Third alternative raises the same problem than the first one whereas the fourth one relates two **Levels** of the subgraph with the same **Cell** (see step 7c). Finally, the last alternative is valid, since we are able to replace `Monthly supply` and `Monthly sales` correspondence by a one-to-many directed context edge -see step 8b- (in fact, they are related by joins raising a many-to-many relationship, but the comparison over the `supplier` field in the WHERE clause turns it into a one-to-many). Furthermore, the **Cells** path do not conform a cycle (step 9); **Cells** at the left side of the one-to-many context edge (i.e. `Monthly supply`) do not select **Measures** (step 10) and there are not alternative branches with one-to-many context edges or nodes each (step 11) either.

Summing up, the algorithm would propose the `Monthly supply`, `Monthly profit` and `Monthly sales` as factual data whereas `Supplier`, `Product` and `Prodtype`, and `Month` would conform the dimensional data.

5.6 Discussion

In this section we have presented a method to validate an end-user multidimensional requirement expressed as an SQL query. In our approach that query is represented by means of a multidimensional graph that lately, is validated as a whole. That is, if we are able to find an implicit multidimensional schema fitting it. With this aim, our work is based on the following criteria:

The cube-query template: We look for that template all over the user query identifying multidimensional concepts. Therefore, it is used to construct the multidimensional graph along steps 1 to 5.

The Base integrity constraint: Levels depicted in the query must identify the multidimensional data. Therefore, they must be orthogonal. It has been used in step 2 to deploy the multidimensional graph, as well as in steps 7a and 7c, in the validation steps.

The correct data summarization integrity constraint: We have followed the three necessary conditions introduced by [LS97] and also introduced in section 2.3:

- Disjointness and Completeness: Used in step 5 to validate the multidimensional conceptual relationships as well as to validate the whole graph (step 6), **Levels** subgraph with regards to the **Cells** placement (step 7d) and the **Cells** path depicted by the context edges (steps 10 to 11). Moreover, the completeness condition has given rise to preserve the multidimensional space when treating with multidimensional relationships allowing zeros (step 5).
- Compatibility: Unfortunately, we have not been able to validate this constraint since that metadata is not captured by the relational schemas. We should ask the user to validate it or ask for a list of valid alternatives.

Therefore, if we can verify that the SQL query given follows the cube-query template; it does not cause summarizability problems and data retrieved is unequivocally identified in the space, we would be able to assure it makes multidimensional sense. Moreover, there are other optional criteria to be used depending on the DW expert:

- **Selection**: If we want to force the user to select data by means of selection comparisons in the WHERE clause, we can validate **Levels** and **Cells** cycles semantics as proposed in steps 7b and 9.
- **Degenerate dimensions**: Multidimensionality is typically modeled in the relational model forcing **Cells** to be related to its analysis **Dimensions**. Therefore, we can assume it when looking for potential multidimensional concepts over the relational databases (obtaining more information). Otherwise, we must bear it in mind as depicted in steps 2 and 4.

6 From Multidimensionality to SQL

This section analyzes the implicit and automatic translation process every ROLAP tool must perform. A preliminary version of this job can be found in [RA05]. Specifically, an end-user would perform navigational and analytical tasks over the organizational data by means of the multidimensional algebra. The set of operations performed in this semantic layer will be automatically processed by the ROLAP tool that will translate it to SQL and therefore, to the relational algebra. The SQL translation of an isolated operation does not represent a problem, but when mixing up the modifications brought about by a set of operations in a single cube-query, some conflicts could emerge depending on the operations involved. Therefore, if these problems are not detected and treated appropriately, the automatic translation can retrieve unexpected results. In this section,

Operation/Source	∅	Selection	Roll-up	Projection	Drill-across	ChangeBase	Union
Selection							
Roll-up	X	X	X		X		
Projection							
Drill-across	X		X		X		
ChangeBase							
Union							

Table 9. Conflicts summary

we define and classify conflicts raised when automatically translating a multidimensional algebra to SQL, and analyze how to solve or minimize their impact.

In section 4 we have presented how an atomic cube-query should be modified when applying an isolated operation over it, but many times end users demand to navigate from **Cube** to **Cube** not just applying isolated operations but performing sequences of operations. Thus, a user chooses a source **Cube** from where starting to operate. Automatically, the ROLAP tool will conform a cube-query to retrieve this **Cube**. Notice this **Cube** is our start point so that it has not been yet manipulated by any operation. Consequently, it is placing a **Cell** of data on the n-dimensional space conformed by its analysis **Dimensions**. This **Cell**, as stated in section 4, could have been materialized or not. If it was, ROLAP tool will retrieve it from an atomic cube-query and if not, it will look for an appropriate **Cell**, in a lower aggregation **Level**, from where obtaining the needed **Cell** by means of **Roll-up**s. For instance, according to figure 1, we could start our analysis from a materialized **Cell** (i.e. `Monthly Profit`) or from a non materialized one (i.e. `Annual Profit`). As `Annual Profit` is not materialized, we need to perform an implicit **Roll-up** over `Monthly Profit` from `Month` to `Year` to get needed data.

As presented in table 9, certain operations may pop up a conflict when combined with an specific *source cube-query*. We refer to a source cube-query as an atomic cube-query modified by a sequence of operations. If no operation has been performed over the atomic cube-query we consider the empty sequence ($\emptyset$). Hence, a cell is crossed ($\times$) when the sequence of operations in the source cube-query contains an specific operation that may cause a conflict with next one to be performed. For instance, it may happen if our source cube-query includes a **Selection** and next operation to be carried out is a **Roll-up**.

Anyhow, any kind of conflict could be avoided using one subquery per multidimensional operation, but we only use subqueries if strictly necessary, shunning the materialization of partial results and easing the RDBMS query optimizer job. Specifically, as presented in [RG03], an important point to note about nested queries (i.e. subqueries) is that a typical optimizer is likely to do a poor job, because of the limited approach to nested queries optimization. In fact, from an efficiency standpoint, they advise us to consider not using nested queries. Main reasons are:

- The nested subquery is fully evaluated in the first step. Consequently, temporal materialization of results may be needed, as well as some good evaluation plans are missed according to the order imposed by nesting. For instance, some proper indexes would never be considered.
- Many times, the query optimizer is not smart enough to find the optimal strategy (for instance, which join algorithm use) and the typical join method used is the "index nested loops" (see [RG03] for more details). Therefore, this approach eludes other join methods that could lead to optimal plans. Moreover, if the nested query is correlated (i.e. a variable from the top-level query also appears in the nested subquery), the nested subquery is evaluated once per outer tuple (i.e. it will be evaluated many times if the correlation field matches many outer tuples).
- If there are several levels of nesting in the query (like in our case if translating each multidimensional operation to one subquery), same approach is considered as presented above just evaluating such queries from the innermost to the outermost and preserving correlation. That is, a correlated subquery may be evaluated once per each high-level query referring to it.

Nevertheless, a nested query often has an equivalent query without nesting, as well as a correlated query often can be many times turned into a decorrelated query. A typical SQL optimizer is likely to find a much better evaluation strategy if it is given the unnested or decorrelated version of the query. However, many of these optimizers are not able to identify that equivalence and transform the initial query to its optimal form.

Notice all conflicts pointed out in table 9 are caused by data aggregation anomalies. In fact, the standard SQL language (and therefore, the relational algebra) will not ever introduce any kind of conflicts. However, as presented in section 3.1, we need to extend the relational algebra in order to support proper data aggregation as demanded in the multidimensional model, giving rise to what is called as the grouping algebra. As introduced in [LS97] and discussed in section 2.3, operations performed must satisfy the disjointness, completeness and compatibility of data handled to guarantee its correct summarization. Otherwise, two operations that, as a whole, do not preserve those three conditions will raise up a conflict. However, since we are trying to avoid subqueries, we need to aggregate in just one SQL query the multiple aggregation of data performed, implicitly or explicitly, by different multidimensional operators.

Therefore, as presented in table 1, **Roll-up** is the only operator performing data aggregation and consequently, it is the only one that may directly raise up conflicts when performed along with other operators. Nevertheless, **Roll-up** is the most important multidimensional operator since it allows us to modify data granularity and for that reason, it is crucial for a ROLAP tool to properly detect and avoid any potential conflict. Specifically, according to table 9, all conflicts are related to **Roll-up** and **Drill-across**. The rest of operations except for **Selection**, propagate conflicts if already present in the cube-query but do not introduce new ones. Consequently, **Projection**, **Union** and **ChangeBase** never raise a conflict. Intuitively, **Projection** removes **Measures** from the SE-

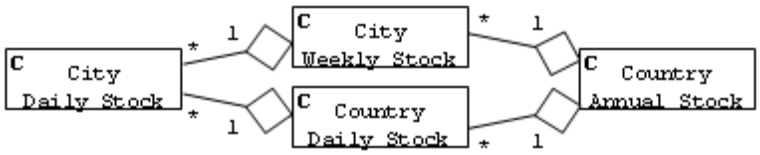

Fig. 12. Example of a hierarchy of Cells

LECT clause and dropping a **Measure** just means to omit a "Cell table" column; **Union** *or*es conditions of two **Cube**s with the same n-dimensional space not removing / adding any point; and **ChangeBase** always asks for a one-to-one relationship in order to be performed, avoiding conflicts due to its own nature. Conversely, **Drill-across** and **Selection** may introduce conflicts in the operators sequence. As presented below, **Drill-across** asks for a one-to-one relationship but sometimes, a one-to-many relationship is enough. In these cases, due to not materialized **Cells**, we need to perform implicit **Roll-ups** to get the necessary one-to-one relationship and being able to raise up the same conflicts caused by a **Roll-up**. Similarly, it may happen with atomic cube-queries not materialized that would need to perform implicit **Roll-ups**. A **Selection** may cause an specific conflict along with a **Roll-up** if we select a subset of points of the **Cube** and later we **Roll-up**, avoiding to take advantage of potential pre-aggregated data. Consequently, notice it is enough to analyze potential conflicts between each pair of operators, since all of them are caused by conciliating multiple aggregations of data in just one SQL query and therefore, order performed among operations does not matter.

Since all conflicts are due to data aggregation anomalies, we have classified conflicts introduced above in three groups according to the three necessary conditions needed to guarantee a correct data summarizability: those performing multiple aggregation functions in a query (not preserving compatibility of data), those due to hidden many-to-many relationships (not preserving disjointness) and finally, those related to the selection granularity (not preserving completeness).

6.1 The Multiple Aggregation Problem

First problem is about functions used to aggregate data. This case typically arises when combining more than one **Roll-up** in the same cube-query. To analyze this problem, we conceptually divide a combination of two **Roll-up**s in two categories depending on whether both were performed over the same **Dimension** or over different ones.

In the first case, we can always solve the problem disregarding first **Roll-up** and just performing the second one, because in a certain moment of time, multidimensional data can only be showed in a certain aggregation **Level** for each **Dimension**. Notice it can always be assumed since, in the worst case, we can

perform a **Roll-up** from the atomic **Level**. Oppositely, when performed over different **Dimension**s we have to compulsory aggregate data for each **Dimension**. Since SQL does not allow us to aggregate data by means of two different functions in the same query this conflict can not be solved in a single cube-query. For instance, if we carry out a **Roll-up** from `Week` to `Year` **Level** in the `Weekly Stock` **Cell**, and later we **Roll-up** from `Year` to **Level** `All`, the whole sequence of both **Roll-up**s can be directly expressed as:

```
SELECT p.ID, "All", c.ID, SUM(s.Stock)
FROM weeklyStock s, Product p, City c
WHERE s.key1 = p.ID AND s.key3 = c.ID
GROUP BY p.ID, c.ID
ORDER BY p.ID, c.ID
```

On the contrary, if we just carry out first **Roll-up**, and later another one from `City` to `Country` along the `Place` **Dimension**, nested queries are compulsory:

```
SELECT p.ID, co.ID, y.ID, SUM(s.Stock)
FROM (SELECT p.ID, c.ID, y.ID, AVG(s.Stock)
    FROM weeklyStock s, Product p,
    City c, Week w, Year y
    WHERE s.key1 = p.ID AND s.key2 = c.ID
    AND s.key3 = w.ID AND w.fkey = y.ID
    GROUP BY p.ID, c.ID, y.ID
    ORDER BY p.ID, c.ID, y.ID), Country co
WHERE s.key1 = p.ID AND s.key2 = c.ID
AND s.key3 = w.ID AND c.fkey = co.ID
GROUP BY p.ID, co.ID, y.ID
ORDER BY p.ID, co.ID, y.ID)
```

Even if SQL allowed us to perform more than one aggregation function in the same query, we would face another problem: the order between aggregation functions. Consider the `Stock` Cell hierarchy detailed in figure 12 extracted from the example presented in figure 1. In this case, `Stock` is analyzed through two **Dimension**s (`Place` and `Time`), and for each possible combination of its **Level**s we got a different **Cell**. For instance, `City Weekly Stock` (containing **cell**s on a `Week-City` granularity **Level**), `Country Annual Stock` (`Country-Year`), `City Daily Stock` (`City-Day`), etc. Thus, it is important to realize that our own multidimensional conceptual design fixes the order of aggregation functions when navigating along **Cell**s hierarchy. If we want to **Roll-up** from `City Daily Stock` to `Country Annual Stock` we have to first aggregate by means of sum (it means, **Roll-up** from `City` to `Country` **Level**) and later aggregate by means of average (**Roll-up** from `Day` to `Year`). So that, order does really matter since sum of averages is different from an average of sums (latter happens when navigating through `City Weekly Stock`). Both orders are possible, but semantics chosen when designing our schema forces us to follow a certain order.

As said, above conflict could be avoided if SQL allowed us to perform more than one aggregation function per query and set up an order between them. For instance, as showed below, an SQL extension stating explicitly two GROUP BY's (very similar to SQL'99 GROUPING SETS modus operandi), would avoid using nested queries when combining more than one conflictive **Roll-up**. First GROUP BY would be related to first aggregation function and analogously to second one:

```
SELECT p.ID, co.ID, y.ID, AVG(SUM(s.Stock))
FROM weeklyStock s, Product p, City c, Week w, Year y, Country co
WHERE s.key1 = p.ID AND s.key2 = c.ID
AND s.key3 = w.ID AND w.fkey = y.ID AND c.fkey = co.ID
GROUP BY p.ID, c.ID, y.ID
GROUP BY p.ID, co.ID, y.ID
ORDER BY p.ID, c.ID, y.ID
```

Although this problem has been presented as a **Roll-up** plus **Roll-up** problem, it goes far beyond as it is crucial when obtaining non materialized **Cell**s from materialized ones. For instance, if we have to work with the `City Weekly Stock` **Cell** that has not been materialized, ROLAP tools will have to perform a **Roll-up** from `Day` to `Week` over `City Daily Stock` to obtain needed data. So that, we have already performed an implicit **Roll-up** that could arise conflicts already presented if we next perform just one explicit **Roll-up**. Similarly, as presented in 6.2, implicit **Roll-up**s can also appear when carrying out a **Drill-across** (also in a **ChangeBase**, but in this case it is raised over the same **Dimension** avoiding any kind of conflict as stated earlier in this section) from a non materialized **Cell**.

Meanwhile, best solution to minimize this problem is to choose with care appropriate **Cell**s to be materialized. An extreme solution would be to materialize all of them, but since it is an exponential space problem, it is not feasible. Hence, in addition to traditional criteria like how frequently would be a **Cell** queried, this problem emphasizes another criterion to decide the usefulness of a given materialized view. According to semantics related to our **Cell**s hierarchy, those **Cell**s whose data can be used as pre-aggregated data to calculate above **Cell**s are good candidates (for instance, in case presented, to materialize `Country Daily Stock` instead of `City Weekly Stock`, since `Country Annual Stock` can only be calculated through the former).

Two possible criterions to decide which **Cell**s materialize could be how frequently would be a **Cell** queried and, according to semantics related to our **Cell**s hierarchy, choose those ones whose data can be used as pre-aggregated data to calculate above **Cell**s (for instance, in case presented, to materialize `Country Daily Stock` instead of `City Weekly Stock`, since `Country Annual Stock` can only be calculated through the former).

6.2 The Fan-Shaped Problem

In this section we introduce a family of problems that are caused because disjointness is not preserved when aggregating data in certain situations. It typically appears related to **Drill-across**, either through semantic relationships or shared **Dimension**s. **Drill-across** asks for a one-to-one relationship, but sometimes a one-to-many relationship is enough. For instance, after dropping the `Place` **Dimension** (by means of **Roll-up** and **ChangeBase**) we can **Drill-across** from `Annual Stock` to `Annual Profit`. Conceptually, the one-to-one relationship is quite clear but in fact, we really have a one-to-many relationship since both **Cell**s are not materialized and `Weekly Stock` and `Monthly Profit` are related to different **Level**s in the `Time` **Dimension**. We can get the needed one-to-one relationship by means of internal **Roll-up**s (from `Month` to `Year` over both

Cells). Since `Year` is not materialized, its descriptors are included along with its children **Level**s in the `Time` **Dimension** hierarchy, given raise to the following query:

```
SELECT p.ID, y.ID, AVG(s.Stock), SUM(m.Profit)
FROM weeklyStock s, monthlyProfit m, Product p
Month mo, Week w, Year y
WHERE m.key1 = p.ID AND m.key2 = mo.ID
AND s.key1 = p.ID AND s.key2 = w.ID
AND mo.yearID = w.yearID
GROUP BY p.ID, y.ID
ORDER BY p.ID, y.ID
```

As enounced in [LS97], the aggregation of data must be disjoint, and in this case, it is not. In fact, what should be a one-to-one relationship turns into a many-to-many one calling up a fan-shaped matching. Thus, we should use a nested query performing first one **Roll-up** and later, the other one, being the "join" last performed. Hence, this problem could be solved if SQL allowed us to state a priority between "joins" and GROUP BY's. However, to minimize its impact it is important, again, to choose with care which **Cell**s should be materialized. Therefore, this is another criterion to bear in mind when deciding the usefulness of a given materialized view.

Finally, also notice that when carrying out a **Drill-across** to a non materialized **Cell**, a ROLAP tool may need to perform internal **Roll-up**s to obtain data to where **Drill-across**. Internal **Roll-up**s followed by an explicit **Roll-up** can cause the same conflict stated in subsection 6.1.

6.3 The Selection Granularity Problem

This problem is closely tied to **Selection** and raises when completeness is not guaranteed. **Selection** allows us to reduce current n-dimensional space by means of a logic clause over a certain **Descriptor**. For instance, selecting those **cell**s of `Daily Stock` related to `Barcelona` in the `Place` **Dimension**. Now, if we **Roll-up** from `Day` to `Week` we cannot change `Daily Stock` to `Weekly Stock` **Cell** in the cube-query to take advantage of pre-aggregated data, since aggregation in `Weekly Stock` is complete and in our current **Cell** it is not (we only have those points related to `Barcelona`). In general, we cannot take advantage of any pre-aggregated data in a materialized **Cell** when translating to SQL if a **Selection** has been carried out over a lower **Level Descriptor** in any of its analysis **Dimension**s. Using appropriate granularity **Cell** and performing internal **Roll-up**s is mandatory. Only way to solve this problem is considering it in the multidimensional schema. For instance, using semantic relationships and creating an specialization of `Daily Stock` (i.e. `Barcelona Daily Stock`) and another on `Weekly Stock` (i.e. `Barcelona Weekly Stock`). Between those **Cell**s, aggregation is complete and we can use the pre-aggregated data without problems.

6.4 Discussion

When analyzing in detail the automatic and implicit translation process a ROLAP tool performs between the multidimensional algebra and SQL, we realize

that there are some additional considerations to be made if we want this process to be free of summarizability problems. Specifically, according to [RG03], it is worth enough to avoid subqueries in order to ease the job of the RDBMS query optimizer, and therefore to conciliate in a single query the SQL translation of multiple multidimensional operators. However, it may embrace to conciliate multiple data summarization in a single query and therefore, we may face potential summarizability problems because of **Roll-up** and not materialized **Cells**.

Since all conflicts depend on summarizability anomalies, **Roll-up** is the only operator that can explicitly cause them. Nevertheless, due to not materialized **Cells**, other operators can raise them by means of implicit **Roll-ups**. Consequently, we have analyzed each possible combination of multidimensional operators to classify these problems. Notice, since each multidimensional operator translation to SQL is embedded in a single query all along with the rest of multidimensional operators performed in the sequence of operations carried out by the end-user, order between operators does not matter at all. That is, analyzing conflicts raised by each pair of multidimensional operators is enough to detect all of them.

We classify those potential summarizability problems according to the three necessary conditions to assure a correct aggregation of data presented in [LS97]; namely: the multiple aggregation problem (not preserving compatibility of data), the fan-shaped problem (not preserving disjointness) and the selection granularity problem (not preserving completeness). Along these problems, we presented how to solve, or at least, smooth them.

Summing up, to guarantee a better performance, these problems must drive the design of the multidimensional schema as well as they must be taken into account when deploying the SQL query in the translation process every ROLAP tool must perform.

7 Conclusions and Future Work

In this paper we have analyzed in detail the mismatch between the multidimensional and the relational model focusing on the implicit translation process a ROLAP tool performs between the multidimensional algebra and SQL (and eventually, to the relational algebra). To do so, we have presented two well-differentiated studies.

First, by means of a conceptual comparative between the multidimensional and the relational algebra, we have remarked the necessity to work in terms of an standard multidimensional algebra. On one hand, we have presented why the relational algebra does not directly fit to multidimensionality. The multidimensional data manipulation should be performed by a restricted subset of the relational algebraic operators; that is, an specific simplification avoiding aggregation problems and defining a closed set of operations. Our main result of this comparative has been the identification of such subset of the relational algebra. On the other hand, we have presented a detailed comparison among the multidimensional algebras introduced in the literature. To the best of our

knowledge, it has been the first comparative about multidimensional algebras carried out. There, we have been able to identify some significant general trends: **Selection**, **Roll-up** and **Drill-down** operators are considered in all the algebras, whereas **Projection**, **Drill-across** and **Union** are included in most of them. Finally, **changeBase** is also considered in the majority of algebras, since most of them agree on the necessity of modifying the n-multidimensional space adding/removing **Dimensions**. Consequently, we strongly believe it could be feasible to agree on a reference multidimensional algebra subsumed by the relational algebra.

Later, we have presented a detailed analysis of the implicit translation a ROLAP tool performs from the multidimensional algebra to SQL. On one hand, we have identified those features a cube-query must enforce to also be semantically correct, analyzing the multidimensional algebra expressiveness with regard to SQL. Based on the criteria that an SQL query must enforce to make multidimensional sense, we have presented an automatic method to validate an SQL as a valid cube-query. Our approach is divided into two main phases: first one creates the multidimensional graph storing relevant multidimensional information about the query, that will facilitate the query validation along the second phase. Such graph represents tables involved in the query and its relationships, and our aim is to label each table as factual data or dimensional data. A correct labeling of all the tables gives rise to a multidimensional schema fitting the input query. Thus, if we are not able to generate any correct labeling, the input query would not make multidimensional sense. Moreover, this approach can be used for multidimensional modeling by examples if we are able to express the multidimensional requirements as SQL queries, as presented in [RA06]. As output, the process will propose, automatically, those multidimensional schemas fitting the input requirements; giving support in the multidimensional design process. On the other hand, we have presented how an atomic cube-query should be modified when applying an isolated operation over it. But many times, end users demand to navigate from **Cube** to **Cube** not just applying isolated operations but performing sequences of operations to be translated in a single cube-query. Certain operations may pop up conflicts due to data aggregation anomalies when combined with an specific source cube-query (a preliminary version of this job can also be found in [RA05]). With this aim, we have classified and analyzed those potential problems in three groups: those performing multiple aggregation functions in a query (not preserving compatibility of data), those due to hidden many-to-many relationships (not preserving disjointness) and finally, those related to the selection granularity (not preserving completeness). We have also presented how to solve or at least smooth these problems avoiding subqueries. These problems have also given rise to two new criteria to decide the usefulness of a given materialized view: according to semantics related to our **Cell**s hierarchy and avoiding hidden many-to-many relationships.

As future work, we will focus on how to conciliate those labeling proposed by our automatic method aimed to validate multidimensional requirements expressed in SQL queries. Given a set of SQL queries, representing each one dif-

ferent multidimensional requirements, the process proposes a set of multidimensional schemas fitting that requirement. With this improvement, the process would also automatically conciliate all those schemas proposed in a set of non-contradictory schemas, conforming an alternative design methodology by examples. Furthermore, one of our future efforts will focus on implementing this method. Finally, next step would consist on generalize those features a cube-query must guarantee to make multidimensional sense, in a generic multidimensional pattern. That pattern would allow us to face an ambitious goal translating it to OWL (Web Ontology Language) or Description Logics and develop a design multidimensional methodology over ontologies representing our organization transactional schemas.

Acknowledgments. This work has been partly supported by the Spanish Ministerio de Educación y Ciencia under project TIN 2005-05406.

References

[Abe02] A. Abelló. *YAM²: A Multidimensional Conceptual Model.* PhD thesis, Universitat Politècnica de Catalunya, 2002.

[AGS97] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling Multidimensional Databases. In *Proc. of the 13th Int. Conf. on Data Engineering (ICDE 1997)*, pages 232–243. IEEE, 1997.

[ASS03] A. Abelló, J. Samos, and F. Saltor. Implementing Operations to Navigate Semantic Star Schemas. In *Proc. of 6th Int. Workshop on Data Warehousing and OLAP (DOLAP 2003)*, pages 56–62. ACM, 2003.

[ASS06] A. Abelló, J. Samos, and F. Saltor. **YAM²** (Yet Another Multidimensional Model): An extension of UML. *Information Systems*, 31(6):541–567, 2006.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. *Database Systems*, pages 65–98, 1972.

[CT97] L. Cabibbo and R. Torlone. Querying Multidimensional Databases. In *Proc. of the 6th International Workshop on Database Programming Languages (DBPL 1997)*, volume 1369 of *LCNS*, pages 319–335. Springer, 1997.

[CT98a] L. Cabibbo and R. Torlone. A Logical Approach to Multidimensional Databases. In *Proc. of 6th Int. Conf. on Extending Database Technology (EDBT 1998)*, volume 1377 of *LNCS*, pages 183–197. Springer, 1998.

[CT98b] L. Cabibbo and R. Torlone. From a Procedural to a Visual Query Language for OLAP. In *Proc. of the 10th Int. Conf. on Scientific and Statistical Database Management (SSDBM 1998)*, pages 74–83. IEEE, 1998.

[FBSV00] E. Franconi, F. Baader, U. Sattler, and P. Vassiliadis. *Fundamentals of Data Warehousing*, chapter Multidimensional Data Models and Aggregation. Springer, 2000. M. Jarke, M. Lenzerini, Y. Vassilious and P. Vassiliadis editors.

[FK04] E. Franconi and A. Kamble. The GMD Data Model and Algebra for Multidimensional Information. In *Proc. of the 16th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2004)*, volume 3084 of *LNCS*, pages 446–462. Springer, 2004.

[GMR98] M. Golfarelli, D. Maio, and S. Rizzi. The Dimensional Fact Model: A Conceptual Model for Data Warehouses. *Int. Journals of Cooperative Information Systems (IJCIS)*, 7(2-3):215–247, 1998.

[HS97] M-S. Hacid and U. Sattler. An Object-Centered Multi-dimensional Data Model with Hierarchically Structured Dimensions. In *Proc. of IEEE Knowledge and Data Engineering Exchange Workshop (KDEX 1997)*. IEEE, 1997.

[HS98] M-S. Hacid and U. Sattler. Modeling Multidimensional Database: A formal object-centered approach. In *Proc. of the 6th European Conference on Information Systems (ECIS 1998)*, 1998.

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the Association for Computing Machinery.*, 29(3):699–717, 1982.

[KRTR98] R. Kimball, L. Reeves, W. Thornthwaite, and M. Ross. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses*. John Wiley & Sons, Inc., 1998.

[Lar99] K.S. Larsen. On grouping in relational algebra. *Int. Journal of Foundations of Computer Science.*, 10(3):301–311, 1999.

[Leh98] W. Lehner. Modelling Large Scale OLAP Scenarios. In *Proc. of 6th Int. Conf. on Extending Database Technology (EDBT 1998)*, volume 1377 of *LNCS*, pages 153–167. Springer, 1998.

[LS97] H.J. Lenz and A. Shoshani. Summarizability in OLAP and Statistical Data Bases. In *Proc. of SSDBM'1997*. IEEE, 1997.

[LW96] C. Li and X.S. Wang. A Data Model for Supporting On-Line Analytical Processing. In *Proc. of 5th Int. Conf. on Information and Knowledge Management (CIKM 1996)*, pages 81–88. ACM, 1996.

[Mic] Microsoft. MDX Specification. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/olapdmad/agmdxbasics_04qg.asp. Last access: 3/24/2006.

[MK00] D.L. Moody and M.A. Kortink. From Enterprise Models to Dimensional Models: A Methodology for Data Warehouse and Data Mart Design. In *Proc. of DMDW'2000*. CEUR-WS.org, 2000.

[ML97] M. and L. Lakshmanan. A foundation for multi-dimensional databases. In *Proc. of 23rd Int. Conf. on Very Large Data Bases (VLDB 1997)*, pages 106–115. Morgan Kaufmann, 1997.

[Ped00] T.B. Pedersen. *Aspects of Data Modeling and Query Processing for Complex Multidimensional Data*. PhD thesis, Faculty of Engineering and Science, 2000.

[PJ01] T.B. Pedersen and C.S. Jensen. Multidimensional Database Technology. *IEEE Computer*, 34(12):40–46, 2001.

[RA05] O. Romero and A. Abelló. Improving Automatic SQL Translation for ROLAP Tools. *Proc. of 9th Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2005)*, 284(5):123–130, 2005.

[RA06] O. Romero and A. Abelló. Multidimensional Design by Examples. In *Proc. of 8th Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK 2006)*. Springer, 2006. To be published in September 2006.

[RG03] R. Ramakrishnan and Johannes Gehrke, editors. *Database Management Systems*. McGraw Hill, 2003.

[TBC99] N. Tryfona, F. Busborg, and J.G.B. Christiansen. starER: A Conceptual Model for Data Warehouse Design. In *Proc. of 2nd Int. Workshop on Data Warehousing and OLAP (DOLAP 1999)*. ACM, 1999.

[TD97] H. Thomas and A. Datta. A Conceptual Model and Algebra for On-Line Analytical Processing in Data Warehouses. In *Proc. of the 7th Workshop on Information Technologies and Systems (WITS 1997)*, pages 91–100, 1997.

[TD01] H. Thomas and A. Datta. A Conceptual Model and Algebra for On-Line Analytical Processing in Decision Support Databases. *Information Systems*, 12(1):83–102, 2001.

[TPGS01] J. Trujillo, M. Palomar, J. Gómez, and I.-Y. Song. Designing Data Warehouses with OO Conceptual Models. *IEEE Computer*, 34(12), IEEE, 2001.

[Vas98] P. Vassiliadis. Modeling Multidimensional Databases, Cubes and Cube operations. In *Proc. of the 10th Statistical and Scientific Database Management (SSDBM 1998)*, pages 53–62. IEEE, 1998.

[Vas00] P. Vassiliadis. *Data Warehouse Modeling and Quality Issues.* PhD thesis, Dept. of Electrical and Computer Engineering (National Technical University of Athens), 2000.

[VS99] P. Vassiliadis and T.K. Sellis. A Survey of Logical Models for OLAP Databases. *SIGMOD Record*, 28(4):64–69, ACM, 1999.

[YP04] X. Yin and T.B. Pedersen. Evaluating XML-extended OLAP queries based on a physical algebra. In *Proc. of 7th Int. Workshop on Data Warehousing and OLAP (DOLAP 2004)*. ACM, 2004.