Multidimensional Design by Examples

Oscar Romero and Alberto Abelló

Universitat Politècnica de Catalunya, Jordi Girona 1-3, E-08034 Barcelona, Spain

Abstract. In this paper we present a method to validate user multidimensional requirements expressed in terms of SQL queries. Furthermore, our approach automatically generates and proposes the set of multidimensional schemas satisfying the user requirements, from the organizational operational schemas. If no multidimensional schema is generated for a query, we can state that requirement is not multidimensional.

Keywords: Multidimensional Design, Design by Examples, DW.

1 Introduction

In this paper we present a method to validate user multidimensional requirements expressed in terms of SQL queries over the organizational operational sources. In our approach, the input query is decomposed to infer relevant implicit and explicit potential multidimensional knowledge contained and accordingly, it automatically proposes the set of multidimensional schemas satisfying those requirements. Thus, facts, dimensions and dimension hierarchies are identified, giving support to the data warehouse design process. Conversely, if our process has not been able to generate any multidimensional schema, we would be able to state that the input query is not multidimensional.

Our main contribution is the automatization of identifying the multidimensional concepts in the operational sources with regard to the end-user requirements. Demand-driven design approaches ([12]) focus on determining the user requirements to later map them onto data sources. This process is typically carried out by the DW expert and it is hardly automatized. Therefore, it is up to the expert criterion to properly point out the multidimensional concepts giving rise to the multidimensional schema. Conversely, in our approach we automatically generate and propose a set of multidimensional schemas validating the input requirements, giving support to the DW expert along the design process.

Notice we propose a method within a supply/demand-driven framework. Our method starts analyzing the requirements stated by the user (in terms of SQL queries), as typically performed in demand-driven approaches. However, it analyzes the operational relational data sources in parallel, as typically performed in supply-driven approaches, to extract additional knowledge needed to validate the user requirements as multidimensional.

We start with section 2 presenting the related work in the literature; section 3 presents the foundations our method is based on whereas section 4 introduces our approach. For the sake of a better comprehension, section 5 presents a practical application of our method and finally, section 6 concludes the paper.

A Min Tjoa and J. Trujillo (Eds.): DaWaK 2006, LNCS 4081, pp. 85–94, 2006.

2 Related Work

As presented in [12], the DW design process can be developed within a supply-driven or a demand-driven approach. Several methodologies following both paradigms have been presented in the literature. On one hand, demand-driven approaches ([12], [5]) focus on determining the user multidimensional requirements (as typically performed in other information systems) to later map them onto data sources. As far as we know, none of them automatize the process. On the other hand, supply-driven approaches ([11], [3], [7], [6] and [2] among others) start thoroughly analyzing the data sources to determine the multidimensional concepts in a reengineering process. In that case, the approach presented in [6] is the only one partially automatizing the process.

As mentioned, our approach combines a demand/supply-driven approach as suggested in [10]. Other works have already combined both approaches, like [5] and [4]. Main difference with our approach is that the first one does not fully automatize the process whereas the second one does not focus on modeling multidimensionality.

3 Framework

In this section we present the criteria our work is based on. That is, those used to validate the input query as a valid multidimensional requirement:

[C1] Relational modeling of multidimensionality: Multidimensionality is based on the fact/dimension dichotomy. Hence, we consider a **Dimension** to contain a hierarchy of **Levels** representing different granularities (or levels of detail) to study data, and a **Level** to contain **Descriptors**. On the other hand, a **Fact** contains **Cells** which contain **Measures**. Like in [11], we consider a **Fact** can contain not just one but several different materialized levels of granularity of data. Therefore, 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 (in the relational model, by means of FK's) to one **Level** for each of its associated **Dimension** of analysis. Finally, one **Fact** and several **Dimension**s to analyze it give rise to a **Star**, to be implemented in the relational model through an "star" or an "snow-flake" schemas as presented in [8].

[C2] The cube-query template: The standard SQL'92 template query to retrieve a **Cell** of data from the RDBMS was first presented in [8]:

```
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, where we can also find logic clauses restricting an specific **Level** attribute (i.e. a **Descriptor**) to a constant $\mathcal{K}$ by means of a comparison operator. 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.

[C3] The Base integrity constraint: Dimensions of analysis should be orthogonal. Despite it could be possible to find **Dimensions** determining others in a multidimensional 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** ([1]). Therefore, we call a **Base** to those minimal set of **Levels** identifying unequivocally a **Cell**, similar to the "primary key" concept in the relational model.

[C4] The correct data summarization integrity constraint: Data summarization performed in multidimensionality must be correct, and we warrant this by means of the three necessary conditions (intuitively also sufficient) introduced in [9]: (1) **Disjointness** (*Sets of* **cell***s at an specific* **Level** *to be aggregated must be disjoint*); (2) **Completeness** (*Every* **cell** *at a certain* **Level** *must be aggregated in a parent* **Level**) and (3) **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. However, this last condition can not be automatically checked unless additional information would be provided, since it is not available neither in the requirements nor in the source schemas.

Multidimensionality pays attention to two main aspects; placement of data in a multidimensional space and summarizability of data. 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 undoubtedly makes multidimensional sense. Moreover, since it is well-known how to model multidimensionality in the relational model, we can look for this pattern over the operational schemas to identify the multidimensional concepts. Additionally, we introduce other optional criteria to validate the query, to be used depending on the DW expert:

[C5] Selections: Multidimensional selections must be carried out by means of logic clauses in the WHERE clause (i.e. *field* comparison operator *constant*). However, we could allow to select data joining two relations through, at least, two different conceptual relationships between them and therefore, not navigating but selecting data equally retrieved by those joins.

[C6] Degenerate dimensions: Multidimensionality is typically modeled forcing **Cells** to be related, by means of FK's, to its analysis **Dimensions** (see [C1]). However, in a non-multidimensional relational schema this may not happen, and we could have a table attribute representing a **Dimension** not pointing to any table (for instance, dates or control numbers). In the multidimensional model, these rather unusual **Dimensions** were introduced in [8], and they are known as "degenerate dimensions".

4 Our Method

Our approach aims to automatically validate a syntactically correct SQL query representing user multidimensional requirements, as a valid (syntactically and semantically) cube-query. An SQL query is a valid cube-query if we are able to generate a non-empty set of multidimensional schemas validating that query. Otherwise, the input query would not represent multidimensional requirements. Multidimensional schemas proposed will be inferred from those implicit restrictions, presented in previous section, an SQL query needs to guarantee to make multidimensional sense; playing the operational databases schemas a key role. This process is divided into two main phases: first one creates what we call the *multidimensional graph*; a graph concisely storing relevant multidimensional information about the query, that will facilitate the query validation along the second phase. Such graph is composed of *nodes*, representing tables involved in the query and *edges*, relating nodes (i.e. tables) joined in the query. Our aim is to label each node as a **Cell** (factual data) or a **Level** (dimensional data). A correct labeling of all the nodes gives rise to a multidimensional schema fitting the input query. Along this section, due to lack of space, we introduce a detailed algorithm in pseudo code to implement our method, followed by a brief explanation of each one of its steps. For the sake of readability, comprehension of the algorithm took priority over its performance:

1. **For each** table **in the** FROM clause **do**
   (a) **Create** a node and **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 Label** $node$ **as Level**;
   (c) **else if** (!degenerate dimensions allowed) **then**
      i. FK = $get\_FK(attribute)$; $node\_dest$ = node; $attributes\_FK$ = attribute;
      ii. while $chain\_of\_FKs\_follows(FK)$ and $FK\_in\_WHERE\_clause(FK)$ **do**
         A. FK = $get\_next\_chained\_FK(FK)$; $node\_dest = get\_node(get\_table(FK))$; $attributes\_FK = get\_attributes(FK)$;
      iii. /* We must also check #attributes selected matches #attributes at the end of the chain. */
      iv. **if** (FK == **NULL** and $\#attrs(attribute) == \#attrs(attributes\_FK)$) **then**
         A. **Label** $node\_dest$ **as Level**;
3. **For each** attribute **in the** SELECT clause **not in the** GROUP BY clause **do**
   (a) node = $get\_node(attribute)$; **Label** node **as Cell with Measures selected**;
4. **For each** comparison **in the** WHERE clause **do**
   (a) node = $get\_node(attribute)$;
   (b) **if** ($!defined\_as\_part\_of\_a\_CK(attribute)$) **then Label** $node$ **as Level**;
   (c) **else if** (!degenerate dimensions allowed) **then**
      i. attribute = $get\_attribute(comparison)$; FK = $get\_FK(attribute)$; $node\_dest = get\_node(attribute)$; $attributes\_FK$ = attribute;
      ii. while $chain\_of\_FKs\_follows(FK)$ and $FK\_in\_WHERE\_clause(FK)$ **do**
         A. FK = $get\_next\_chained\_FK(FK)$; $node\_dest = get\_node(get\_table(FK))$; $attributes\_FK = get\_attributes(FK)$;
      iii. **if** (FK == **NULL** and $\#attributes(attribute) == \#attributes(attributes\_FK)$) **then**
         A. **Label** $node\_dest$ **as Level**;
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) $multiplicity = get\_multiplicity(set\_of\_joins)$; relationships fitting = {};
   (d) **For each** relationship **in** $get\_allowed\_relationships(multiplicity)$ **do**
      i. **if** ($!contradiction\_with\_graph(relationship)$) **then**
         A. relationships fitting = relationships fitting + {relationship};
   (e) **if** ($!sizeof(relationshipsfitting)$) **then** return $notify\_fail$("Tables relationship not allowed");
   (f) **Create** an edge($get\_join\_attributes(set\_of\_joins)$); **Label** edge **to** relationships fitting;
   (g) **if** ($unequivocal\_knowledge\_inferred(relationships\_fitting)$) **then** propagate knowledge;

Table 1. Valid multidimensional relationships in a relational schema

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	✓	✓	×	×

The algorithm starts analyzing each query clause according to [C2]:

Step 1: Each table in the FROM clause is represented as a node in the multidimensional graph. Along the whole process we aim to label them and, if in a certain moment, an already labeled node is demanded to be labeled with a different tag, the process ends and raises the contradiction stated.

Step 2: The GROUP BY clause must fully functionally determine data (see [C3]). Thus, fields on it represent dimensional data. However, we can not label them directly as **Levels** since, because of [C1], **Cells** are related to **Levels** by FK's and dimensional data selected could be that in the **Cell** table. Hence, we label it as a **Level** if that field 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. Then, the table where the FK's chain ends plays a **Level** role. If [C6] is assumed, we can not rely on FK's to point out **Levels**.

Step 3: Those aggregated attributes in the SELECT not present in the GROUP BY surely play a **Measure** role. Hence, each node is labeled as a **Cell** with selected **Measures**. If the input query does not contain a GROUP BY clause, we are not forced to aggregate **Measures** by means of aggregation functions in the SELECT, and this step would not be able to point them out.

Step 4: Since a multidimensional **Selection** must be carried out over dimensional data, this step labels nodes as **Levels** with the same criteria regarding FK's presented in step 2.

Step 5: Previous steps are aimed to create and label nodes whereas this step creates and labels edges. For each join between tables in the WHERE clause, 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 multiplicity, we look for those allowed multidimensional relationships depicted in table 1, not contradicting previous knowledge in the graph. If we find any, we create an edge representing that join and label it with those allowed relationships. Finally, if we are just considering one possible 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 updated.

Next, we need to validate the graph 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 non-contradictory labeling alternatives, to be validated each one as follows:

```
6. If !connected(graph) then return notify_fail("Aggregation problems because of cartesian product.");
7. For each subgraph of Levels in the multidimensional graph do
   (a) 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 return notify_fail("Cycles
            can not be used to select data.");
       iii. else ask user for semantical validation;
   (b) if exists_two_Levels_related_same_Cell(subgraph) then return notify_fail("Non-orthogonal
       Analysis Levels");
   (c) For each relationship in get_1_to_N_Level_Level_relationships(subgraph) do
         i. if left_related_to_a_Cell_with_Measures(relationship) then return
            notify_fail("Aggregation Problems.");
8. For each Cell pair in the multidimensional graph do
   (a) For each 1_1_correspondence(Cellpair) do Create context edge between Cell pair;
   (b) For each 1_N_correspondence(Cellpair) do Create directed context edge between Cell pair;
   (c) If exists_other_correspondence(Cellpair) then return notify_fail("Invalid correspondence
       between Cells.");
9. if contains_cycles(Cells path) then
   (a) if contradiction_about_paths_multiplicities(Cells path) then return notify_fail("Cycles can
       not be used to select data.");
   (b) else ask user for semantical validation; 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 return notify_fail("Aggregation
    problems among Cells.");
```

Step 6: The multidimensional graph must be connected to avoid the "Cartesian Product" ([C6]). Moreover, it must be composed of valid edges giving rise to a path among **Cells** (factual data) and connected subgraphs of **Levels** (dimensional data) surrounding it.

Step 7: This step validates **Levels** subgraphs with regard to **Cells** placement: According to [C3], two different **Levels** in a subgraph can not be related to the same **Cell** (step 7b); to preserve [C4], **Level - Level** edges raising aggregation problems on **Cells** with **Measures** selected must be forbidden (step 7c), and finally, if we do not consider [C5], every subgraph must represent a valid **Dimension** hierarchy (i.e. not being used to select data). Thus, we must be able to point out two nodes in the subgraph representing the *top* and *bottom* **Levels** of the hierarchy, and if there are more than one alternative path between those nodes, they must be semantically equivalent (7a).

Step 8: **Cells** determine multidimensional data and they must be related somehow in the graph. Otherwise, they would not retrieve a single **Cube** of data. For every two **Cells** in the graph, we aim to validate those paths between them as a whole, inferring and validating the multiplicity raised as follows: (1) 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** (i.e. a context edge replaces that subgraph representing the one-to-one correspondence). As depicted in figure 1.1, 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**. (2) 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. Finally, many-to-many relationships between **Cells** would invalidate the graph since they do not preserve disjointness.

Steps 9, 10 and 11: Previous step has validated the correspondences between **Cells** whereas these steps validate the **Cells** path (multidimensional data retrieved) as a whole: According to [C5], step 9 validates cycles in the path of **Cells** to assure they are not used to select data, similar to the **Levels** cycles

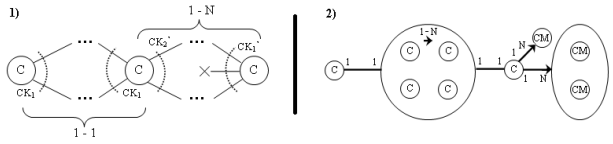

Fig. 1. Examples of **Cells** paths of context edges and nodes

validation. Once the cycle has been validated, **Cells** involved are clustered in a *context node* labeled with the cycle multiplicity, as showed in figure 1.2. Steps 10 and 11, according to [C4], look for potential aggregation problems. First one looks for **Cells** with **Measures** selected at the left side of a one-to-many context edge or node whereas second one looks for alternative branches with one-to-many context edges or nodes each, raising a forbidden many-to-many relationship between **Cells** involved (as depicted in figure 1.2).

5 A Practical Example

In this section, we present a practical example of the method introduced along this paper. We consider figure 2 (where CK's are underlined and FK's dash-underlined) to depict part of the operational schema of the organization. Therefore, given the following requirement: *"Retrieve benefits obtained with regard to supplier* `'ABC'`*, per month"*, it could be expressed 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
```

We aim to decide if this query makes multidimensional sense. If it does, our method will propose the set of multidimensional schemas satisfying our multidimensional needs. First, we start constructing the multidimensional graph. In our case, we do not consider degenerate dimensions (see [C6]):

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

Step 2 and 3: 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 up to the `Supplier` node, where the FK chain ends. Consequently, we label the `Supplier` node as a **Level**.

Finally, 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

```
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. 2. The organizational relational database schema

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. 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 1 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.

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 3.

To validate the graph, first, we check if the graph is connected (in this case, it is). Next, since some nodes have not been labeled, we find out all the valid alternatives by means of backtracking. 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 non-contradictory labeling alternatives depicted in table 2 (those crossed out are eluded in this step since they raise up contradictions).

For each labeling retrieved by the backtracking algorithm, we try to validate the graph. For instance, we will follow in detail the validation algorithm for the first alternative, where all three unknown nodes are labeled as **Cells**:

- We validate each subgraph of **Levels** (namely those isolated **Levels** depicted in figure 3) with regard to **Cells**. Since they do not contain cycles (alternative paths) of **Levels**; there is neither two **Levels** in the same subgraph related to the same **Cell** nor forbidden **Level** - **Level** relationships, both are correct.
- Next, 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`

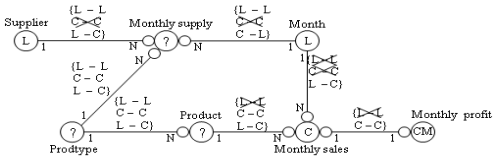

Fig. 3. The multidimensional graph deployed

unique correspondence (through the `Month` node) can not be replaced by a context edge (they are only linked through their **Month** field; i.e. joining two pieces of CK's and raising a forbidden many-to-many context edge).

Since we have found a contradiction, we elude this labeling and try the next one. Second labeling is forbidden because it raises a one-to-many **Level** - **Level** relationship (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**. 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 (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; **Cells** at the left side of the one-to-many context edge (i.e. `Monthly supply`) do not select **Measures**, and there are not alternative branches with one-to-many context edges or nodes each either.

Summing up, the algorithm would conclude that requirement is multidimensional and 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.

6 Conclusions

Based on the criteria that an SQL query must enforce to make multidimensional sense, we have presented a method to validate multidimensional requirements expressed in terms of an SQL 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

Table 2. Labeling alternatives retrieved

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	×

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 fulfilling the requirements expressed in the input query. Thus, if we are not able to generate any correct labeling, the input query would not represent multidimensional requirements. As future work, we will focus on how to conciliate those labeling proposed by our method for different multidimensional requirements.

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

References

1. A. Abelló, J. Samos, and F. Saltor. **YAM**^{2} (Yet Another Multidimensional Model): An extension of UML. *Information Systems*, 31(6):541–567, 2006.
2. M. Böhnlein and A. Ulbrich vom Ende. Deriving Initial Data Warehouse Structures from the Conceptual Data Models of the Underlying Operational Information Systems. In *Proc. of 2nd Int. Workshop on Data Warehousing and OLAP (DOLAP 1999)*, pages 15–21. ACM, 1999.
3. 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.
4. D. Calvanese, L. Dragone, D. Nardi, R. Rosati, and S. Trisolini. Enterprise Modeling and Data Warehousing in TELECOM ITALIA. *Information Systems*, 2006.
5. P. Giorgini, S. Rizzi, and M. Garzetti. Goal-oriented requirement analysis for data warehouse design. In *Proc. of 8th Int. Workshop on Data Warehousing and OLAP (DOLAP 2005)*, pages 47–56. ACM Press, 2005.
6. 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.
7. B. Hüsemann, J. Lechtenbörger, and G. Vossen. Conceptual Data Warehouse Modeling. In *In Proc. of DMDW'00)*. CEUR-WS.org, 2000.
8. 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.
9. H.J. Lenz and A. Shoshani. Summarizability in OLAP and Statistical Data Bases. In *Proc. of SSDBM'1997*. IEEE, 1997.
10. S. Luján-Mora and J. Trujillo. A comprehensive method for data warehouse design. In *In Proc. of DMDW'2003*, volume 77. CEUR-WS.org, 2003.
11. 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.
12. R. Winter and B. Strauch. A Method for Demand-Driven Information Requirements Analysis in Data Warehousing Projects. In *In Proc. of HICSS'03*, pages 231–239. IEEE, 2003.