Using Design Guidelines to Improve Data Warehouse Logical Design

Verónika Peralta, Raúl Ruggia
Instituto de Computación, Universidad de la República. Uruguay.
{vperalta,ruggia}@fing.edu.uy

Abstract. Data Warehouse (DW) logical design techniques often start from a conceptual schema and then generate the relational structures. Applying this approach involves to cope with two main aspects: (i) mapping the conceptual model structures to the logical model ones, and (ii) taking into account implementation issues, which are not included in the conceptual schema. Existing work in this area has mainly focused in the first aspect and proposes inter-model mapping strategies that lead to different relational structures. The second aspect has been less studied. This paper presents a set of *design guidelines* to state such implementation issues. These guidelines enable to cope with several design problems, for example: managing complex and big dimensions, dimension versioning, different user profiles accessing to different attributes, high summarized data, horizontal partitions of historical data, generic dimensionality and non-additive measures. This work is part of a DW logical design environment, where the design guidelines are specified through a graphical editor and then automatically processed in order to build the logical schema.

Keywords. Data Warehouse design, logical design methods.

1 Introduction

It is widely accepted that Data Warehouse logical design is significantly different to OLTP database design [17][18][2][8][9]. Relational DWs are designed to optimize the execution of queries and to provide simple and expressive meanings to express these queries [18][2]. Such requirements have lead to the proposition of specific design techniques and patterns, like the well-known Star, Snowflake and Star Cluster Schemas [18][23].

Many of the existing DW design techniques propose the construction of a DW conceptual schema and the generation of the DW logical schema from the conceptual one [13][6][3][15][29]. This is a traditional database design strategy [4][21] and presents two main aspects: (i) mapping the conceptual model structures to the logical model ones, and (ii) taking into account implementation issues, which are not included in the conceptual schema. While the first aspect copes with defining the relational structures to be created, the second one enables to obtain a schema that satisfies performance and maintenance requirements as well as the information retrieval ones.

Most of the previously referenced techniques take into account the first aspect, and propose inter-model mapping strategies that lead to different DW-oriented relational structures. The second aspect has been less studied. In [13] the authors propose to take as input DW workload information (the data volumes of tables, a set of user queries and their execution frequency). This information is used to complement a mapping strategy in order to obtain a better logical DW schema.

It is important to note that dealing with additional, implementation issues is not straightforward. Firstly, it is not possible to formalize all the real world implementation-related aspects. Second, if these specifications are intended to be automatically treatable, then they have to be simpler enough to do not rely in undecidable problems. Finally, it should be practical to obtain and manage such specifications.

This paper addresses the problem of obtaining a DW logical schema from a conceptual one, and presents a set of *design guidelines* to declare such implementation issues. It follows a similar approach to [13], but proposes to use implementation-related guidelines instead of workload information in order to obtain a more accurate logical schema.

The proposed *design guidelines* enable to state: (i) the desired degree of fragmentation for dimension tables, supporting different DW styles and strategies like managing big dimensions, versioning and considering using user profiles, (ii) the degree of fragmentation of fact tables, allowing to manage tables with fewer records and improve query performance and considering user profiles and strategies to manage historical data, and (iii) the aggregated data to materialize, considering addtivity issues and trying to obtain a balance between performance and storage.

We believe that they constitute a well-balanced mechanism, because includes very synthetic implementation-related information that enables to obtain accurate logical schemas in several design cases, and it is automatically treatable as well as practical to be specified by a designer.

The *design guidelines* are part of a more general DW design environment, which automates most of the DW logical schema generation [27][28][7].

The main contribution of this paper is the proposition of a set of design guidelines, which allow expressing implementation-related information in a simple way and are used to generate accurate DW relational schemas through a semi-automated process.

The rest of this paper is organized as follows: Section 2 studies related work and discusses some problems in the mappings between multidimentional and relational models. Section 3 presents the design guidelines and discusses criteria for their definition. Section 4 present the DW logical design environment and section 5 concludes.

2 Relational Representations for Multidimensional Structures

This section presents a brief state of the art of relational DW design proposals that start from conceptual multidimensional schemas, and discusses some problematic cases that would arise in pure mapping techniques.

2.1 Existing techniques to relational DW design

There are several proposals concerning the generation of DW relational schemas from conceptual ones [13][6][3][15][29]. They mainly deal with DW conceptual design, but present mapping strategies (conceptual → logical), and generate DW relational schemas following fixed design strategies.

In [6], the MD conceptual model is presented and two algorithms are proposed in order to show that conceptual schemas can be mapped to relational or multidimensional logical models. The mapping to a relational schema is straightforward and generates a star schema.

In [13] a methodological framework, based in the DF conceptual model is proposed. Starting from a conceptual model, they propose to generate relational or multidimensional schemas. Despite not suggesting any particular model, the star schema is taken as example. They also present a design strategy based on a cost model that takes as input the DW query workload and data volumes, and determine how to vertically fragment fact tables to materialize them [14][20]. Query workload is stated as pairs <query, frequency> and it is based on the conceptual structures. Although it is a powerful approach, it does not cover table maintenance considerations (e.g. related to manage versioned *monster dimensions* [18]) , and does not state how to determine the most relevant query workload pairs.

Other proposals [3][15][29] generate fixed star schemas. As these proposals do not aim at generating complex DW structures, they do not offer flexibility to apply different design strategies.

Other works in DW logical design do not take a conceptual schema as input to design the logical schema [23][5][18]. They directly build logical models either from requirements or from source databases.

In [18] the author proposes to build star schemas from user requirements and presents logical design techniques to solve frequent DW design problems.

In [23], the logical schema is built from an Entity-Relationship (E/R) schema of the source database. It describes several logical models, as star, snowflake and star-cluster, and presents the main characteristics of each one. The paper proposes two operators: collapse hierarchy and aggregation, and states how to derive the different logical models applying the operators. The designer has to decide when to apply each operator and which model to obtain.

The proposition of [5] also builds a logical schema starting from a source database.

2.2 Motivating examples

In this section we present an example and several situations to show some limitations of existing approaches. This motivates the use of additional information to take into account implementation issues, which are not specified in the conceptual schema.

Along the paper we use the following conceptual schema to illustrate the examples.

Example 1. Consider the case of a company that brings phone support to its customers and wants to analyze the amount of time spent in call attentions.

In the conceptual design phase, the DW analyst has identified two dimensions: *customers* and *dates*. The *customers* dimension has four levels: *state, city, customer* and *department*, organized in two hierarchies. The *dates* dimen-

sion has two levels: *year* and *month*. The designer also identified a fact: *support* that crosses *customers* and *dates* dimensions and takes the call *duration* as measure.

Figure 1 sketches the conceptual schema using CMDM graphical notation [7]. We use the CMDM conceptual model, but the examples can be stated with other conceptual models.

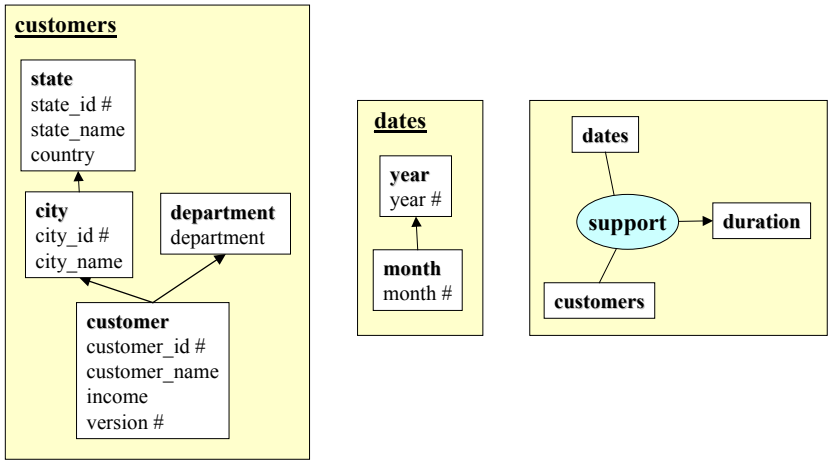

Figure 1. Conceptual schema. Dimension representation consists of levels in hierarchies, which are stated as boxes with their names in bold followed by the items. The items followed by a sharp (#) identify the level. Arrows between levels represent dimension hierarchies. Facts are represented by ovals linked to the dimensions, which are stated as boxes. Measures are distinguished with arrows.

The proposals for generating a DW relational schema from a conceptual schema mainly focus on the generation of star schemas [13][6][15]. Although Star Schema is the most popular DW relational pattern, it does not adequately model relevant situations that lead to fragment dimension and fact tables. As a consequence, other alternatives have been proposed, for example normalizing part of the dimensions [23], materializing aggregates [18] and fragmenting fact schemas [14].

It is interesting to note that, although existing proposals are strongly based in particular conceptual models, the resulting tables are quite similar. They consist of a fact table for each conceptual fact and a dimension table for each conceptual dimension. The following tables (S1) are generated from the conceptual schema of Example 1 in order to build a star schema:

```
(S1) CUSTOMERS (customer_id, department, city_id, state_id, customer_name,
          income, version, city_name, state_name, country)
     DATES (month, year)
     SUPPORT (month, customer_id, minutes)
```

We identify several problems when following a fixed strategy that always builds stars schemas:

Firstly, in the case of complex or big dimensions (monster dimensions [18]), denormalization can cause great redundancy, and therefore maintenance, problems. Partially normalizing the dimension can be a trade-off between performance and redundancy. Consider that the CUSTOMERS table has hundreds of thousands customers, but only one hundred different cities. Data about cities and states is extremely redundant and causes maintainability problems. The following tables can be more appropriate:

```
(S2) CUSTOMERS (customer_id, department, city_id, customer_name, income,version)
     CITIES (city_id, state_id, city_name, state_name, country)
```

If the dimension data has to be versioned (to keep track of the changes), denormalization can cause great maintainability problems. It is useful to maintain different tables for attributes that do not change, attributes that slowly change and attributes that change more frequently. For example, if we want to trace the history of the cities where a customer has lived, we can add additional tuples to the CUSTOMERS table and generalize the key (we can use the version attribute to do it). Table will increase size and degrade unnecessarily the performance of other queries, e.g. by

the department attribute. To avoid this, we can store the current city in the CUSTOMERS table and the historical values in a separate table with generalized key, as in S3:

```
(S3) CUSTOMERS (customer_id, department, city_id, customer_name, income,version)
     CUSTOMER_HISTORY (customer_id, version,city_id)
     CITIES (city_id, state_id, city_name, state_name, country)
     SUPPORT (month, customer_id, version, minutes)
```

Furthermore, when there are requirements to access different subsets of dimension attributes, it is not necessary to design large and complex dimension tables. Consider that there are two user profiles: those who are mainly interested in the geographical distribution of calls, and those who supervise customers by their departments. The fragmentation of the customer dimension in two tables: CUSTOMERS and CITIES as in (S2) or (S3) is recommended.

Another problem arises when frequent queries require high-summarized data. Query execution time is generally not satisfactory and we need to materialize aggregated data. Consider for example these two types of requirements: (i) analyze calls of the last months, filtering by customers, their cities and departments, and (ii) analyze annual totals of previous years calls for each city. To solve these requirements the schema should materialize the following aggregation:

```
(S4) SUPPORT_YEAR_CITY (year, city_id, minutes)
```

In addition, when materializing an aggregate, additional dimension tables (with the appropriate granularity) must be generated to assure correct results. These can be an additional reason to partially normalize a dimension table. E.g. to join the SUPPORT_YEAR_CITY table with city and state information we need the CITIES table of schema S2. If city and state data is also stored in the denormalized CUSTOMERS table of schema S1, we have excessive duplication of data. In this situation, schema S2 is more suitable.

Sometimes, some dimension attributes can be more frequently accessed than others. Consider that the *state_name* attribute is queried in most of the queries, but the *country* attribute is only queried in some specific ones. We can store different degrees of redundancy for the different attributes, storing some of them in several tables, for example, storing the *state_name* attribute in both dimension tables:

```
(S5) CUSTOMERS (customer_id, department, city_id, customer_name, income,
          version, state_name)
     CITIES (city_id, state_id, city_name, state_name, country)
```

Usually, the most frequent queries access the most recent data, and other queries access older data. In such cases, fact tables can be horizontally fragmented (and eventually aggregated) according to the user queries. Consider that calls of the last months are queried grouped by customer, and calls of previous years are queried grouped by year and city. We can build two tables: a fact table only with calls of previous years and aggregated by year and city; and another fact table with the current year calls without aggregation.

```
(S6) CURRENT_SUPPORT (month, customer_id, version, minutes)
     HISTORICAL_SUPPORT (year, city_id, minutes)
```

Other aggregates may be necessaries because of complex requirements. Consider the analysis of the quantity of customers that make long calls, classifying call durations in different ranges. In this case, the *duration* measure is used as dimension, and the *customer* dimension is studied as a measure. This is a case of *generic dimensionality* [9] and is a hard query. It may be necessarily to materialize the aggregation:

```
(S7) CUSTOMER_QUANTITY (month, city_id, duration_range, quantity)
```

In the generated schema (S7), the *customer quantity* measure is not additive, for example for the *dates* dimension. If we have the customer quantity for the different months of a year, we cannot sum these quantities to obtain the value corresponding to the year because some customers can be counted several times. We have basically two types of solutions: querying the detailed fact table asking for *distinct* customers (with possible performance problems), and materializing several aggregates for the most important crossings. For example, we can materialize annual totals

(S8). Queries with additivity problems are generally not considered when selecting which aggregates to materialize. But they must be taken into account because they must be correctly solved even if they are not the most frequently executed. Additivity issues are discussed in [19].

```
(S8) | ANUUAL_CUSTOMER_QUANTITY (year, city_id, duration_range, quantity)
```

Finally, taking into account all the presented implementation-oriented considerations, a DW designer should build the following relational schema:

```
(S)  | DATES (month, year)
     | CUSTOMERS (customer_id, department, city_id, customer_name, income,
     |      version, state_name)
     | CUSTOMER_HISTORY (customer_id, version,city_id)
     | CITIES (city_id, state_id, city_name, state_name, country)
     | CURRENT_SUPPORT (month, customer_id, version, minutes)
     | HISTORICAL_SUPPORT (year, city_id, minutes)
     | CUSTOMERQUANTITY (month, city_id, duration_range, quantity)
     | ANNUALCUSTOMERQUANTITY (year, city_id, duration_range, quantity)
```

There are notorious differences between schema S and the one generated by the Star Schema based methodology (S1). The differences are a consequence of taking into account more information than the pure mapping methodology.

Then, it can be concluded that implementation related information is required to obtain accurate relational DW schemas. Design methods need this kind of information in order to generate schemas that provide efficient access to data, easy maintenance of data, and efficient use of storage resources.

At this point arise some questions:

- What kind of implementation-related information has to be stated to obtain an accurate relational DW schema?
- How do DW design techniques should make use of these statements?

The next sections address these issues.

3 Design Guidelines

We propose a set of design guidelines to expresses design decisions related to non-functional requirements as performance, storage constraints and use of the system.

The use of guidelines constitutes a flexible way to express design strategies and properties of the DW in a high-level way. This allows the application of different design styles and techniques, generating the DW logical schema following the designer approach.

We propose guidelines related to *Aggregate Materialization*, *Horizontal Fragmentation of Facts* and *Vertical Fragmentation of Dimensions*. We think that with these three guidelines we achieve a trade-off between the expressiveness and simplicity of the mechanism.

3.1 Aggregate materialization

During conceptual design, the analyst identifies the desired facts, which leads to the implementation of fact tables at logical design. These fact tables can be stored with different degrees of detail, i.e. maximum detail tables and aggregate tables. In the example, for the *support* fact of Example 1 we can store a fact table with detail by customers and months, and another table with totals by departments and years.

Giving a set of levels of the dimensions that conform the fact, we specify the desired degree of detail to materialize it.

We define a structure called *cube* that basically specifies the set of levels of the dimensions that conform the fact. These levels indicate the desired degree of detail for the materialization of the cube.

Sometimes, they do not contain any level of a certain dimension, representing that this dimension is totally summarized in the cube. However, the set can contain several levels of a dimension, representing that the data can be

summarized by different criteria from the same dimension. A cube may have no measure, representing only the crossing between dimensions (factless fact tables [18]).

A cube is a 4-uple <Cname, R, Ls, M> where *Cname* is the cube name that identifies it, *R* is a conceptual schema fact, *Ls* is a subset of the levels of the fact dimensions and *M* is an optional measure that can be either an element of *Ls* or null.

The SchCubes set, defined by extension, indicates all the cubes that will be materialized (see Definition 1).

- SCHCUBES $\subseteq$ { <CNAME, R, LS, M> /
  CNAME $\in$ STRINGS $\wedge$
  R $\in$ SCHFACTS $\wedge$
  LS $\subseteq$ { L $\in$ GETLEVELS(D) / D $\in$ GETDIMENSIONS(R) } $\wedge$
  M $\in$ (LS $\cup \perp$)
  }[1]

Definition 1 – Cubes. A cube is formed by a name, a fact, a set of levels of the fact dimensions and an optional measure. SCHCUBES is the set of cubes to be materialized.

Figure 2 shows the graphical notation for cubes. There are two cubes: detail and summary of the *support* fact of Example 1. Their levels are: *month, customer and duration,* and *year*, *city and duration*, respectively. Both of them have duration as measure.

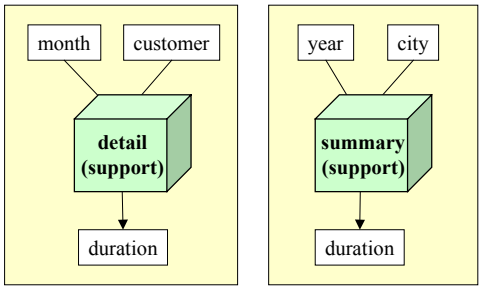

Figure 2. Cubes. They are represented by cubes, linked to several levels (text boxes) that indicate the degree of detail. An optional arrow indicates the measure. Inside the cube are its name and the fact name (between brackets).

3.2 Horizontal fragmentation of facts

A cube can be represented in the relational model by one or more tables, depending on the desired degree of fragmentation.

To horizontally fragment a relational table is to build several tables with the same structure and distribute the tuples between them. In this way, we can store together the tuples that are queried together and have smallest tables, which improve query performance.

As an example, consider the *detail* cube of Figure 2 and suppose that most frequent queries correspond to calls performed after year 2002. We can fragment the cube in two parts, one to store tuples from calls after year 2002 and the other to store tuples from previous calls. The tuples of each fragment must verify respectively:

- month $\geq$ January-2002
- month < January-2002

The horizontal fragmentation in distributed databases has been studied in [24]. They define two properties (completeness and disjointness) that must be verified by a horizontal fragmentation in order to assure completeness while minimizing the redundancy. A fragmentation is *complete* if each tuple of the original table belongs to one of the

[1] SCHFACTS is the set of facts of the conceptual schema. The functions GETLEVELS and GETDIMENSIONS return the set of levels of a dimension and the set of dimensions of a fact respectively.

fragments, and it is *disjoint* if each tuple belongs to only one fragment. As an example, consider the following fragmentation:

- month ≥ January-2002
- month ≥ January-1999 ∧ month < January-2001
- month < January-2000

The fragmentation is not complete because tuples corresponding to calls of year 2001 do not belong to any fragment, and it is not disjoint because tuples corresponding to calls of year 1999 belongs to two fragments.

In a DW context, these properties are important to be considered but they are not necessarily verified. If a fragmentation is not *disjoint* we will have redundancy that is not a problem for a DW system. For example, we can store a fact table with all the history and a fact table with the calls of the last year. Completeness is generally desired at a global level, but not necessarily for each cube. Consider, for example, the two cubes of Figure 2. We define a unique fragment of the *detail* cube (with all the history) and a fragment of the *summary* cube with the tuples after year 2002. Although the *summary* cube fragmentation is not complete, the history tuples can be obtained from the *detail* cube performing a roll-up, and there is no loose of information.

We define a fragmentation as a set of strips, each one represented by a predicate over the cube instances.

- SCHSTRIPS ⊆ { <SNAME, C, PRED> /
  SNAME ∈ STRINGS ∧
  C ∈ SCHCUBES ∧
  PRED ∈ PREDICATES(GETITEMS(C))
  }[2]

Definition 2 – Strips. A strip is formed by a name, a cube and a boolean predicate expressed over the items of the cube levels.

Figure 3 shows the graphical notation for strips. There are two strips for the *detail* cube: *current* and *history*, for calls posteriors and previous to 2002, respectively.

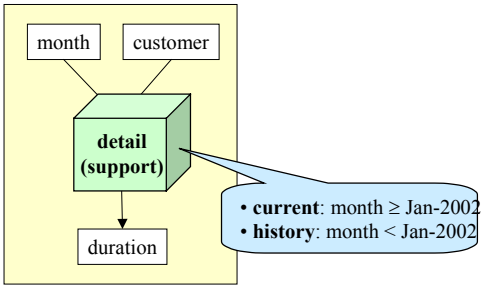

Figure 3 – Strips. The strip set for a cube is represented by a callout containing the strip names followed by their predicates.

3.3 Vertical fragmentation of dimensions

Through this guideline, the designer specifies the level of normalization he wants to obtain in relational structures for each dimension. In particular, he may be interested in a star schema, denormalizing all the dimensions, or conversely he may prefer a snowflake schema, normalizing all the dimensions [23]. This decision can be made globally, regarding all the dimensions, or specifically for each dimension. An intermediate strategy is still possible by indicating the levels to be stored in the same table, allowing more flexibility for the designer. To sum up, for specifying this guideline, the designer must indicate which levels of each dimension he wishes to store together. Each set of levels is called a *fragment*.

Given two levels of a fragment (A and B) we say that they are *hierarchically related* if they belong to the same hierarchy or if there exists a level C that belongs to two hierarchies: one containing A and the other containing B. A

[2] PREDICATES(A) is the set of all possible boolean predicates that can be written using elements of the set A. The GETITEMS function returns the set of items of the cube levels.

fragment is valid only if all its levels are hierarchically related. For example, consider a fragment of the *customers* dimension of Example 1 containing only *city* and *department* levels. As the levels do not belong to the same hierarchy, storing them in the same table would generate the cartesian product of the levels' instances. However, if we include the *customer* level to the fragment, we can relate all levels, and thus the fragment is valid.

In addition, the fragmentation must be *complete*, i.e. all the dimension levels must belong to at least one fragment to avoid losing information. If we do not want to duplicate the information storage for each level, the fragments must be disjoint, but this is not a requirement. The designer decides when to duplicate information according to his design strategy.

- SCHFRAGMENTS $\subseteq$ { <FNAME, D, LS> /
  FNAME $\in$ STRINGS $\wedge$
  D $\in$ SCHDIMENSIONS $\wedge$
  LS $\subseteq$ GETLEVELS(D) $\wedge$
  $\forall$A,B $\in$ LS . (
  <A,B> $\in$ GETHIERARCHIES(D) $\vee$
  <B,A> $\in$ GETHIERARCHIES (D) $\vee$
  $\exists$ C $\in$ LS . (<A,C> $\in$ GETHIERARCHIES (D) $\wedge$ <B,C> $\in$ GETHIERARCHIES (D)))

}[3]

Definition 3 – Fragments. A fragment is formed by a name, a dimension and a sub-set of the dimension levels that are hierarchically related.

Graphically, a fragmentation is represented as a coloration of the dimension levels. The levels that belong to the same fragment are bordered with the same color. Figure 4 shows three alternatives to fragment the *customers* dimension.

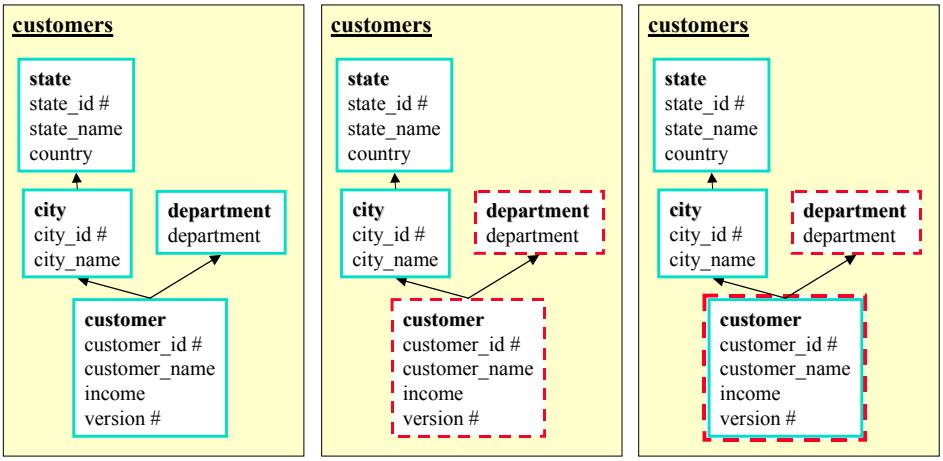

Figure 4. Alternative fragmentations of *customers* dimension. The first one has only one fragment with all the levels. The second alternative has two fragments, one including *state* and *city* levels (continuous line), and the other including the *customer* and *department* levels (dotted line). The last alternative keeps the *customer* level in two fragments (dotted and continuous line).

3.4 Criteria for the definition of guidelines

By means of the guidelines, the designer defines:

- A set of cubes for each fact.
- A set of strips for each cube.
- A set of fragments for each dimension.

[3] SCHDIMENSIONS is the set of dimensions of the conceptual schema. The GETHIERARCHIES function returns the pairs of levels that conform the dimension hierarchies [7].

In order to specify the guidelines, the designer should consider performance and storage constraints as well as use and maintenance requirements.

As for each fact, a set of cubes is specified, then, it is reasonable to materialize the cube with lower granularity to do not loose information, and the cubes that improve performance for the most frequent or complex queries. It is not feasible to materialize all possible cubes, then, the decision is a trade-off between storage and performance. Queries with additivity problems must be also considered, despite of the query access frequencies, because sometimes, summary data cannot be obtained from detailed data.

To perform horizontal fragmentations of a cube, the designer has to study two factors: table size and the subset of tuples that are frequently queried together. The more strips the designer defines, the lower size they will have, and there will be better response time for querying them, but queries that involve several strips are worsen. The decision must be based on the requirements, looking at which data is queried together.

The manner to fragment the dimensions is related to the chosen design pattern. The definition of fragments with several levels (denormalization) achieves better query response time but increases redundancy. If dimension data changes slowly, redundancy is not a problem. But if dimension data changes frequently and we want to keep the different versions, the maintenance cost grows. This guideline tries to find a balance between query response time and redundancy. Data that is not queried together is a good indicator in order to fragment the dimension.

4 The DW logical design environment

In this section we present an environment for DW logical design.

Our underlying design methodology for DW logical design takes as input the source database and a conceptual multidimensional schema.

Then, the design process consists of three main tasks:

(i) *Refine the conceptual schema* adding design guidelines and obtaining a "*refined conceptual schema*".

(ii) *Map the refined conceptual schema to the source database*. The mapping indicates how to calculate each multidimensional structure from the source database [27].

(iii) *Generate the relational DW schema according to the refined conceptual schema and the source database.* For the generation we propose a rule-based mechanism in which the DW schema is built by successive application of transformations to the source schema [22]. Each rule determines which transformation must be applied according to the design conditions given by the refined conceptual schema, the source database and the mappings between them, i.e., when certain design condition is fulfilled, the rule applies certain transformation [28].

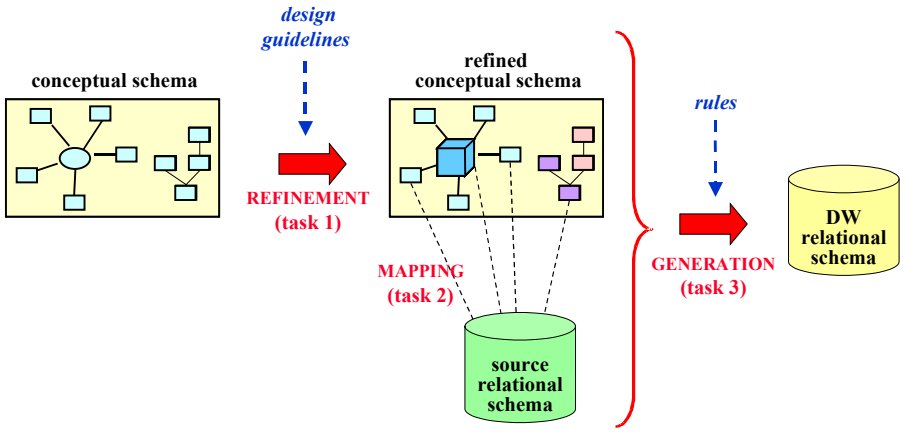

Figure 5. DW logical design environment. Guidelines refine the conceptual schema (task 1). Mappings indicate how to calculate each conceptual structure from the source database (task 2). The refined conceptual schema, the source database and the mappings between them are the input to the automatic rule-based mechanism that generates the DW relational schema.

For example, to generate an aggregation table several tasks are performed: the conceptual schema express the fact, guidelines specify the detail level for the aggregation, mappings indicate how to calculate each attribute of the

aggregation from the source database, and the generation process applies the appropriate transformations to build the DW relational schema transforming the source database.

Figure 5 shows the proposed environment.

This framework has been prototyped [26]. The system applies the rules and automatically generates the logical schema by executing an algorithm that considers the most frequent design problems suggested in existing methodologies and complemented with practical experiences. The prototype also includes a user interface for the definition of guidelines and mappings, an environment for the interactive application of the rules and the schema transformations and a graphical editor for the CMDM conceptual model. All this work has been implemented in a CASE environment for DW design.

5 Conclusions

Data Warehouse logical design involves defining relational structures that satisfy information requirements as well as implementation related ones (i.e. performance and maintainability). While the first kind of requirements is represented by means of the conceptual schema, the second one lack on most of existing methodologies.

This paper presented a notation for specifying implementation related requirements *(design guidelines)*, which enables to complement the conceptual schema with declarations of *Aggregate Materializations*, *Horizontal Fragmentation of Facts* and *Vertical Fragmentation of Dimensions*.

The proposed guidelines enable to cope with several design problems, for example: managing complex and big dimensions, dimension versioning, different user profiles accessing to different attributes, high summarized data, horizontal partitions of dimensions with aggregates, generic dimensionality, non additive measures. The set of design guidelines does not intend to be complete, hence it should be extended and complemented with other proposals, like [13].

By using these guidelines, the design methodology can generate more accurate relational DW schemas than the ones generated applying only model mapping strategies.

The presented design guidelines are used in the context of a CASE environment. They are specified through a graphical editor and then processed in order to build the logical schema. These tools are currently being developed in the context of projects of the CSI Group [10].

Current work consists in the extension of the framework to support multiple source databases, and to manage all the metadata with a CWM [24] repository. In the near future we intend to enhance the capabilities of our framework, integrating further design techniques and extending it to cover the design of loading and refreshment tasks.

References

[1] Abello, A.; Samos, J.; Saltor, F.: "A Data Warehouse Multidimensional Data Models Clasification". Technical Report LSI-2000-6. Dept. Lenguages y Sistemas Informáticos, Universidad de Granada, 2000.

[2] Adamson, C.; Venerable, M.: “Data Warehouse Design Solutions”. J. Wiley & Sons, Inc.1998.

[3] Ballard, C.; Herreman, D.; Schau, D.; Bell, R.; Kim, E.; Valncic, A.: “Data Modeling Techniques for Data Warehousing”. SG24-2238-00. IBM Red Book. ISBN number 0738402451. 1998.

[4] Batini, C.; Ceri, S.; Navathe, S.: “Conceptual Database Design- an Entity Relationship Approach”. Benjamin Cummings, 1992.

[5] Boehnlein, M.; Ulbrich-vom Ende, A.:”Deriving the Initial Data Warehouse Structures from the Conceptual Data Models of the Underlying Operational Information Systems". DOLAP’99, USA, 1999.

[6] Cabibbo, L. Torlone, R.:"A Logical Approach to Multidimensional Databases", EDBT'98, Spain, 1998.

[7] Carpani, F. Ruggia, R.: “An Integrity Constraints Language for a Conceptual Multidimensional Data Model”. SEKE’01, Argentina, 2001.

[8] Chaudhuri, S.; Dayal, U.: "*An overview of Data Warehousing and OLAP technology*". SIGMOD Record, 26(1), 1997.

[9] Codd, E.F.; Codd, S.B.; Salley, C.T.: "Providing OLAP (on-line analytical processing) to user- analysts: An IT mandate". Technical report, 1993.

[10] CSI Group, Universidad de la República, Uruguay. URL: http://www.fing.edu.uy/inco/grupos/csi/

[11] Elmasri, R.; Navathe, S.: “Fundamentals of Database Systems. 2nd Edition”. Benjamin Cummings, 1994.

[12] Golfarelli, M.; Maio, D.; Rizzi, S.:"Conceptual Design of Data Warehouses from E/R Schemes.", HICSS'98, IEEE, Hawaii,1998.

[13] Golfarelli, M. Rizzi, S.: "Methodological Framework for Data Warehouse Design.", DOLAP'98, USA, 1998.

[14] Golfarelli, M. Maio, D. Rizzi, S.:"Applying Vertical Fragmentation Techniques in Logical Design of Multidimensional Databases". DAWAK'00, UK, 2000.

[15] Hahn, K.; Sapia, C.; Blaschka, M.: "Automatically Generating OLAP Schemata from Conceptual Graphical Models", DOLAP'00, USA, 2000.

[16] Hüsemann, B.; Lechtenbörger, J.; Vossen, G.:"*Conceptual Data Warehouse Design*". DMDW'00, Sweden, 2000.

[17] Inmon, W.: "Building the Data Warehouse". John Wiley & Sons, Inc. 1996.

[18] Kimball, R.:"The Datawarehouse Toolkit". John Wiley & Son, Inc., 1996.

[19] Lenz, H. Shoshani, A.: "Summarizability in OLAP and Statistical Databases". Conf. on Statistical and Scientific Databases, 1997.

[20] Maniezzo, V.; Carbonaro, A.; Golfarelli, M.; Rizzi, S.: "ANTS for Data Warehouse Logical Design". *4th Metaheuristics International Conference*, Porto, pp. 249-254, 2001.

[21] Markowitz, V. Shoshani, A.: "On the Correctness of Representing Extended Entity-Relationship Structures in the Relational Model". SIGMOD'89, USA, 1989.

[22] Marotta, A. Ruggia, R.: "Data Warehouse Design: A schema-transformation approach". SCCC'2002. Chile. 2002.

[23] Moody, D.; Kortnik, M.: "From Enterprise Models to Dimensionals Models: A Methodology for Data Warehouse and Data Mart Design". DMDW'00, Sweden, 2000.

[24] Object Management Group: "The Data Warehousing, CWM and MOF Resource Page". URL: http://www.omg.org/cwm/

[25] Ozsu, M.T. Valduriez, P.: "Principles of Distributed Database Systems". Prentice-Hall Int. Editors. 1991.

[26] Peralta, V.: "Diseño lógico de Data Warehouses a partir de Esquemas Conceptuales Multidimensionales". Technical Report. Universidad de la República, Uruguay. TR-0117. 2001.

[27] Peralta, V.; Marotta, A.; Ruggia, R.: "Towards the Automation of Data Warehouse Design". Technical Report. Universidad de la República, Uruguay. TR-0309. 2003.

[28] Peralta, V.; Illarze, A.; Ruggia, R.: "On the Applicability of Rules to Automate Data Logical Design". DSE'03, Velden, Austria, 2003.

[29] Phipps, C.; Davis, K.: "Automating data warehouse conceptual schema design and evaluation". DMDW'02, Canada, 2002.