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 often start with a conceptual schema and then generates relational structures. Applying this approach implies 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 considered in the conceptual schema. This paper addresses this second aspect and presents a formalism that allows the DW designer to specify *design guidelines* which express design strategies related with implementation requirements. Through these guidelines the designer states high level manners to cope with different 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 Relational Data Warehouse logical design is significantly different from OLTP database design [17][18][2][8][9]. Relational DWs are designed to provide simple and expressive meanings to complex queries as well as to optimize their execution [18][2]. Such requirements have lead to specific design techniques and patterns, like the well-known Star, Snowflake and Star Cluster Schemas [18][23].

Many of the existing DW logical design techniques propose to build the DW logical schema from a conceptual schema [13][6][3][15][29]. This is a traditional database design strategy [4][21] which involves two main issues: (i) map the conceptual model structures to the logical model ones, and (ii) take into account implementation-oriented requirements, which are not considered in the conceptual schema. While the first issue is related with the definition of the relational structures to be created, the second one concerns the construction of a schema that satisfies performance and maintenance requirements. Most of the previously referenced techniques address the first aspect by proposing 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 to complement a mapping strategy.

It is important to note that dealing with implementation-oriented requirements is not straightforward. Firstly, it is not possible to formalize all the real world implemen-

tation-related aspects. Second, if these specifications are intended to be automatically processed, then they have to be simpler enough to avoid undecidable problems. Finally, the mechanism to obtain and to manage the specifications should be practical.

This paper addresses the problem of obtaining a DW logical schema from a conceptual one, and presents a formalism to declare implementation-related requirements which may be used in automated logical design methods. The paper follows a similar approach to [13], but proposes the specification of implementation-related *guidelines* instead of workload information in order to obtain an accurate logical schema.

The proposed formalism consists of three types of *design guidelines*: (i) aggregate materializations, (ii) horizontal fragmentation of facts (or cubes), (iii) vertical fragmentation of dimensions. These *design guidelines* enable the designer to formally state some kinds of implementation related characteristics. For example: the degree of fragmentation for dimension tables, the degree of fragmentation of fact tables, and the materialization of aggregated data. The proposed guidelines are also simple enough to be processed by an automated tool.

The declaration of *design guidelines* is part of a wider DW design environment, which intends to automate most of the DW logical schema generation [27][28][7].

The main contribution of this paper is the proposition of a formalism that allows expressing implementation-related guidelines in a simple way and which can be used to generate accurate DW relational schemas through 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 multidimensional and relational models. Section 3 presents the formalism to declare 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 first a brief state of the art on relational DW design techniques that take as input conceptual multidimensional schemas, and then discusses some problematic cases that would arise when building the DW relational schemas.

2.1 Existing techniques to relational DW design

There are several proposals addressing the generation of DW relational schemas from conceptual ones [13][6][3][15][29].

In [6], the MD conceptual model is introduced and two algorithms are proposed in order to map conceptual schemas to either relational or multidimensional logical models. The former one is straightforward generating a star schema. In [13], a methodological framework based on the DF conceptual model is proposed. The paper proposes to generate relational or multidimensional schemas starting from a conceptual one. Despite not suggesting any particular model, the star schema is taken as example. The paper also presents a design strategy based on a cost model that takes the DW query workload and data volumes as input, and states the criteria to build vertical fragments of fact tables in order to materialize them [14][20]. Query workload is stated as pairs <query, frequency> and it is based on the conceptual structures. Al-

though 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 focus on star schemas and are not conceived to generate complex DW structures [3][15][29]. Therefore they do not provide flexibility to apply different design strategies.
Other works in DW logical design do not take a conceptual schema as input, but build the logical schema either from requirements or the source databases [23][5][18]. 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 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 proposition of [5] also builds a logical schema starting from a source database.

2.2 Motivating examples

This section presents a couple of examples that show potential problems that may arise when the relational DW schema is build using a fixed logical design strategy. Such limitations motivate the specification of additional information to take into account implementation-related aspects in the DW logical design stage. The examples will be based on the following situation.

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

The conceptual design phase leads to two dimensions: *customers and, dates*. *Customers* dimension has four levels: *state, city, customer* and *department*, organized in two hierarchies. *Date* dimension has two levels: *year* and *month*. The designer has also identified a set of facts associated to events of attention *support*, which crosses *customers* and *dates* dimensions, and which includes a measure representing the *Duration* of an attention. Note that this measure (*Duration)* can also be seen as a dimension (by applying the generic dimensionality principle of Codd). Consequently it is structured into two levels: *minutes* and *ranges*.

Figure 1 sketches the conceptual schema using CMDM graphical notation [7], but can be used with other conceptual models.

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) correspond to the Example 1, and follow a Star Schema pattern:

```
(S1) CUSTOMERS (customer_id, department, city_id, state_id,
          customer_name, income, 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 star schemas.

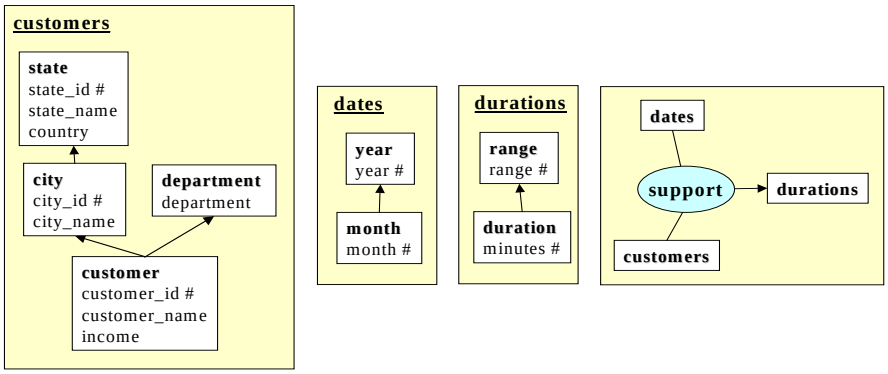

Figure 1. Conceptual schema. Dimension representation consists of levels in hierarchies, which are stated as boxes with their names in bold followed by the items. The items followed by a sharp (#) identify the level. Arrows between levels represent dimension hierarchies. Facts are represented by ovals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

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

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

In addition, if the dimension data has to be versioned (to keep track of the changes), again denormalization can cause maintainability problems. A good strategy would be to maintain different tables gruping 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 add the version attribute to do it). Then, the size of the table will increase and the performance of other queries, e.g. by the department attribute, will degrade. To avoid this, we can store the current city in the CUSTOMERS table and the historical values in a separate table with a 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. In order to satisfy these requirements the schema should materialize the following aggregation (in addition to the SUPPORT table of S3):

```
(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 may 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 may be accessed more frequently than others. Consider that the *state_name* attribute is queried in most of the queries, but the *country* attribute is queried in only 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 first one generated by the Star Schema based methodology (S1). The differences are a consequence of taking into account implementation-related information .

Therefore, it can be concluded that implementation related information is required to obtain accurate relational DW schemas. Furthermore, design methods and tools should use 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 some questions arise:

- 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 A Formalism to specify Design Guidelines

We propose a formalism to represent information related to non-functional requirements, which provides guidelines to perform the relational DW logical design. This formalism consists of the so-called *design guidelines*, which are of three types: *Ag-*

gregate Materialization, *Horizontal Fragmentation of Facts* and *Vertical Fragmentation of Dimensions*.

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, 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 allows the designer to declare the degree of detail for the materialization of each fact. A cube basically specifies the set of levels of the dimensions that conform the fact.

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).

$$
\begin{aligned}
&\textsc{SchCubes} \subseteq \{\ \langle \textsc{Cname},\ R,\ \textsc{Ls},\ M\rangle\ /\\
&\quad \textsc{Cname} \in \textsc{Strings} \wedge\\
&\quad R \in \textsc{SchFacts} \wedge\\
&\quad \textsc{Ls} \subseteq \{\ L \in \textsc{GetLevels}(D)\ /\ D \in \textsc{GetDimensions}(R)\ \} \wedge\\
&\quad M \in (\textsc{Ls} \cup \bot)\\
&\}^{1}
\end{aligned}
$$

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.

[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.

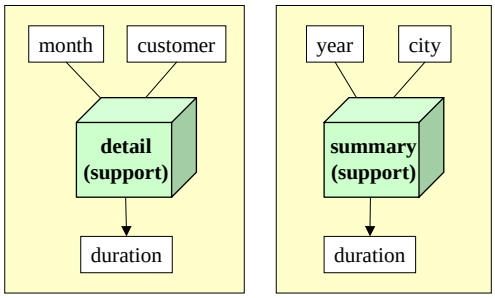

Figure 2. Cubes. They are represented by cubes, linked to several levels (text boxes) that indicate the degree of detail. An optional arrow indicates the measure. The cube name and the fact name (between brackets) are inside the cube.

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. Horizontal fragmentation of relational tables is a well known technique, which leads to smaller tables and to improve query performance [25].

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 ≥ “January-2002”
- month < “January-2002”

In order to ensure completeness while minimizing the redundancy, horizontal fragmentation has to satisfy two properties: completeness and disjointness. A fragmentation is *complete* if each tuple of the original table belongs to one of the 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 structure called *strip* that allows the designer to declare how to fragment a cube. A fragmentation is expressed 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.

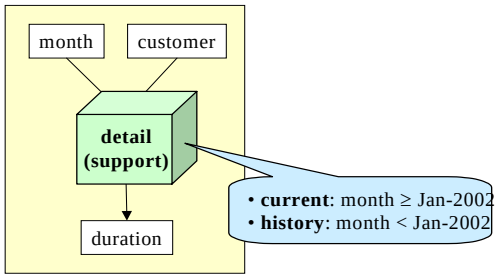

Figure 3 – Strips. The strip set for a cube is represented by a callout containing the strip names followed by their predicates.

3.3 Vertical fragmentation of dimensions

The guideline called *fragment* allows the designer to specify which levels of each dimension he wishes to store together. A fragment basically consists of a set of levels of a dimension. This may lead to star schema, denormalizing all the dimensions, or conversely he may prefer a snowflake schema, normalizing all the dimensions [23], or may choose intermediate options. This decision can be made globally, regarding all the dimensions, or specifically for each dimension.

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 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 gen-

[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.

erate 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.

SCHFRAGMENTS ⊆ { <FNAME, D, LS> /
FNAME ∈ STRINGS ∧
D ∈ SCHDIMENSIONS ∧
LS ⊆ GETLEVELS(D) ∧
∀A,B ∈ LS . (
<A,B> ∈ GETHIERARCHIES(D) ∨
<B,A> ∈ GETHIERARCHIES(D) ∨
∃C ∈ LS . (<A,C> ∈ GETHIERARCHIES(D) ∧ <B,C> ∈ 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.

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.

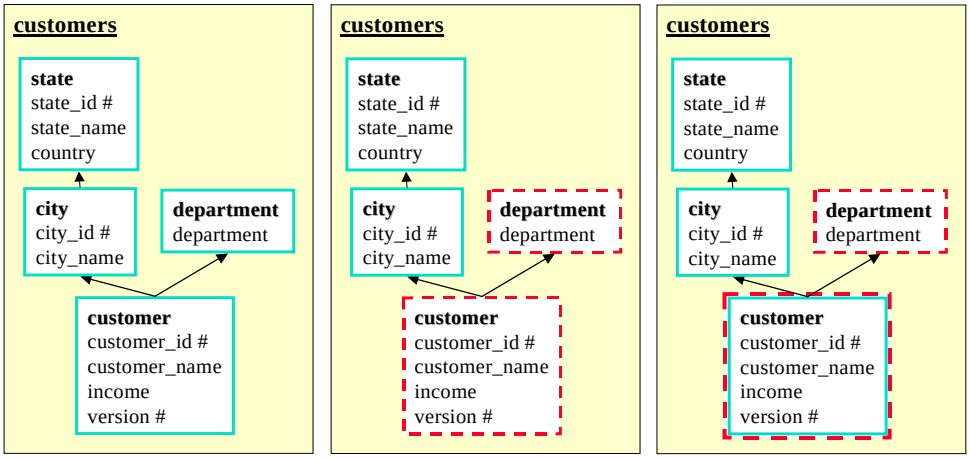

Figure 4. Alternative fragmentations of *customers* dimension. The first one has only one fragment with all the levels. The second alternative has two fragments, one including *state* and *city* levels (continuous line), and the other including the *customer* and *department* levels (dotted line). The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

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

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 (or pattern). Figure 4 shows three alternatives to fragment the *customers* dimension.

3.4 Criteria specifying the 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.

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

It is not feasible to materialize all possible cubes for a fact, then, the decision is a trade-off between storage and performance constraints. It is reasonable to materialize the cube with the lowest granularity to do not loose information, and the cubes that improve performance for the most frequent or complex queries. 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 then the 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 takes as input the source database and a conceptual multidimensional schema. Then, the design process consists of three main tasks:

- *Refine the conceptual schema* adding design guidelines and obtaining a "*refined conceptual schema*".
- *Map the refined conceptual schema to the source database*. The mappings indicate how to calculate each multidimensional structure from the source database [27].

- *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].

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.

Figure 5 shows the proposed environment.

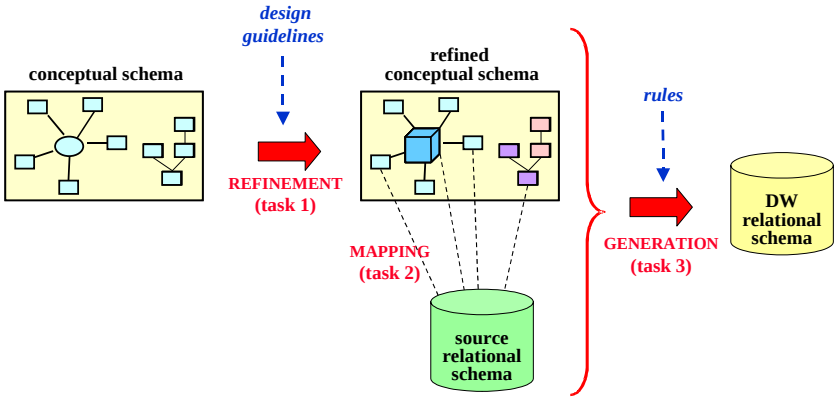

Figure 5. DW logical design environment. Guidelines refine the conceptual schema (task 1). Mappings indicate how to calculate each conceptual structure from the source database (task 2). The refined conceptual schema, the source database and the mappings between them are the input to the automatic rule-based mechanism that generates the DW relational schema (task 3).

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 formalism 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 use of guidelines constitutes a flexible way to express design strategies and properties of the DW in a high-level manner. This allows the application of different

design styles and techniques, generating the DW logical schema following the designer approach. Furthermore, these three guidelines we achieve a trade-off between the expressiveness and the property of being automatically processed. Although the proposed guidelines enable to cope with several design problems, the set of design guidelines does not intend to be complete; hence it should be extended and complemented with other proposals like [13].

The aforementioned design guidelines have been implemented in the context of a CASE environment in the context of projects of the CSI Group [10]. Current work consists in the extension of the environment to support multiple source databases, and to manage all the metadata with a CWM [24] repository.

References

[1] Abello, A.; Samos, J.; Saltor, F.: "A Data Warehouse Multidimensional Data Models Clasification". Technical Report LSI-2000-6. 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, Austria, 2003.

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