Towards the Automation of Data Warehouse Design

Verónika Peralta, Adriana Marotta, Raúl Ruggia
Instituto de Computación, Universidad de la República. Uruguay.
vperalta@fing.edu.uy, amarotta@fing.edu.uy, ruggia@fing.edu.uy

Abstract. Data Warehouse logical design involves the definition of structures that enable an efficient access to information. The designer builds relational or multidimensional structures taking into account a conceptual schema representing the information requirements, the source databases, and non-functional (mainly performance) requirements. Existing work in this area has mainly focused into aspects like data models, data structures specifically designed for DW, and criteria for defining table partitions and indexes. This paper proposes a step forward on the automation of DW relational design through a rule-based mechanism, which automatically generates the DW schema by applying existing DW design knowledge. The proposed rules embed design strategies which are triggered by conditions on requirements and source databases, and perform the schema generation through the application of predefined DW design oriented transformations. The overall framework consists of: the rules, design guidelines that state non-functional requirements, mapping specifications that relate the conceptual schema with the source database schema, and a set of schema transformations.

Keywords. Data Warehouse design, Rule based design, Schema transformations.

1 Introduction

A Data Warehouse (DW) is a database that stores information devoted to satisfy decision-making requests. DWs have features that distinguish them from transactional databases [9][21]. Firstly, DWs are populated from source databases, mainly through batch processes [6] that implement data integration, quality improvement, and data structure transformations. In addition, the DW has to support complex queries (grouping, aggregates, crossing of data) instead of short read-write transactions (which characterizes OLTP). Furthermore, end users analyze DW information through a multidimensional perspective using OLAP tools [10][20].

Adopting the terminology of [4][11][1], we distinguish three different design phases: *conceptual design* manages concepts that are close to the way users perceive data; *logical design* deals with concepts related to a certain kind of DBMS (e.g. relational, object oriented,) but are understandable by end users; and *physical design* depends on the specific DBMS and describes how data is actually stored. In this paper we deal with logical design of DWs.

DW logical design processes and techniques differ from the commonly applied on the OLTP database design. DW considers as input not only the conceptual schema but also the source databases, and is strongly oriented to queries instead of transactions. Conceptual schemas are commonly specified by means of conceptual multidimensional models [1]. Concerning the logical relational schema, various data structures have been proposed in order to optimize queries [21]. These structures are the well-known star schema, snowflake, and constellations [21][3][23].

An automated DW logical design mechanism should consider as input: a conceptual schema, non-functional requirements (e.g. performance) and mappings between the conceptual schema and the source database. In addition, it should be flexible enough to enable the application of different design strategies. The mentioned mappings should provide the expressions that define the conceptual schema items in terms of the source data items. These mappings are also useful to program the DW loading and refreshment processes.

This paper addresses DW logical (relational) design issues and proposes a rule-based mechanism to automate the generation of the DW relational schema, following the previously described ideas.

Some proposals to generate DW relational schemas from DW conceptual schemas are presented in [15][7][3][17]. We believe that existing work lack in some main aspects related to the generation and management of mappings to source databases, and flexibility to apply different design strategies that enable to build complex DW structures (for example, explicit structures for historical data management, dimension versioning, or calculated data).

The here proposed mechanism is based on rules that embed logical design strategies and take into account three constructions: (1) a *multidimensional conceptual schema*, (2) *mappings*, which specify correspondences between the elements of the conceptual and source schemas, and (3) *guidelines*, which provide information on DW logical design criteria and state non functional requirements. The rules are triggered by conditions in these constructions, and perform the schema generation through the application of predefined DW design oriented transformations.

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. This flexibility is a very important quality for an automatic process.

The main contribution of this paper is the specification of an open rule-based mechanism that applies design strategies, builds complex DW structures and keeps mappings with the source database.

The remaining sections of this paper are organized as follows: Section 2 discusses related work. Section 3 presents a global overview of the proposition and an example. Section 4 describes the design guidelines and the mappings between the DW conceptual schema and the source schema. Section 5 presents the rule-based mechanism, describes the DW schema generation algorithm and illustrates it within an example. Finally, Section 6 points out conclusions and future work.

2 Related work

There are several proposals concerning to the automation of some tasks of DW relational design [15][7][3][17][23][5].

Some of them mainly focus on conceptual design [15][7][3][17] and deal with the generation of relational schemas following fixed design strategies. In [7], the MD conceptual model is presented (despite authors calling it a logical model) and two algorithms are proposed in order to show that conceptual schemas can be translated to relational or multidimensional logical models. But the work does not focus on a logical design methodology. In [15] 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. The translation process is left to the designer, but interesting strategies and cost models are presented. Other proposals [3][17] also generate fixed star schemas.

As the proposals do not aim at generating complex DW structures, they do not offer flexibility to apply different design strategies. Furthermore, they do not manage complex mappings, between the generated relational schemas and the source ones, though some of them derive the conceptual schema from the source schema.

Other proposals do not take a conceptual schema as input to the logical design task [23][5][21]. In [23], the logical schema is built from an Entity-Relationship (E/R) schema of the source database. Several logical models, as star, snowflake and star-cluster are studied. In [21] logical design techniques are presented by means of examples.

Other works related to automated DW design mainly focus in DW conceptual design and not logical design, like [18][26] which generate the conceptual schema from an E/R schema of the source database.

3 The DW logical design environment

In this section we present the proposed environment for DW logical design and we illustrate our approach within an example. The approach is described in a more detailed way in the following sections.

3.1 General description

Our underlying design methodology for DW logical design takes as input the source database and a conceptual multidimensional schema ([14][7][18][26][8]).

Then, the design process consists of three main tasks:

(i) **Refine the conceptual schema** adding non functional requirements and obtaining a "*refined conceptual schema*". The conceptual schema is refined by adding *design guidelines* that express non-functional requirements. As an example of *guideline*, the designer indicates that he wants to fragment historical data or states the desired degree of normalization.

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

(iii) **Generate the relational DW schema according to the refined conceptual schema and the source database**. For the generation of the DW relational schema 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.

Figure 1 shows the proposed environment.

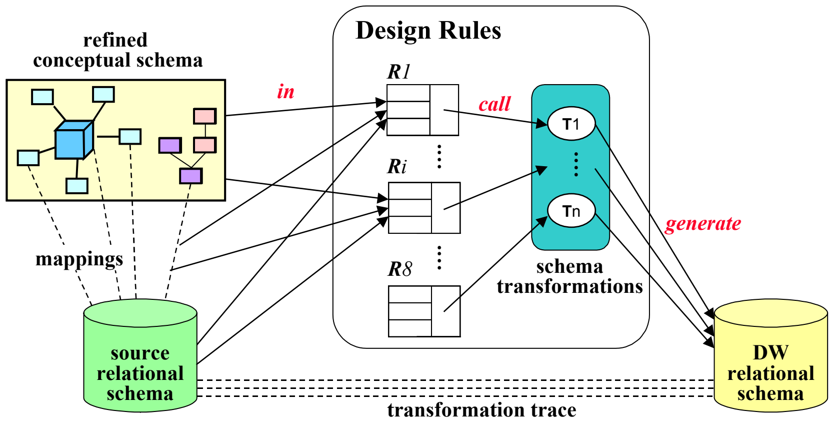


Figure 1. DW logical design environment. The refined conceptual schema, the source database and the mappings between them (left side) are the input to the automatic rule-based mechanism that builds the DW relational schema (right side) applying schema transformations.

The environment provides the infrastructure to carry out the specified process. It consists of:

- A *refined conceptual schema*, which is built from a conceptual multidimensional schema enriched with design guidelines.
- The *source schema* and the *DW schema*.
- Schema *mappings*, which are used to represent correspondences between the conceptual schema and the source schema.
- A set of *design rules*, which apply the *schema transformations* to the source schema in order to build the DW schema.
- A set of pre-defined *schema transformations* that build new relations from existing ones, applying DW design techniques.
- A *transformation trace*, which keeps the transformations that where applied, providing the mappings between source and DW schemas.

3.2 Motivation example

Consider a simple example 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* dimension 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 2 sketches the conceptual schema using CMDM graphical notation [8].

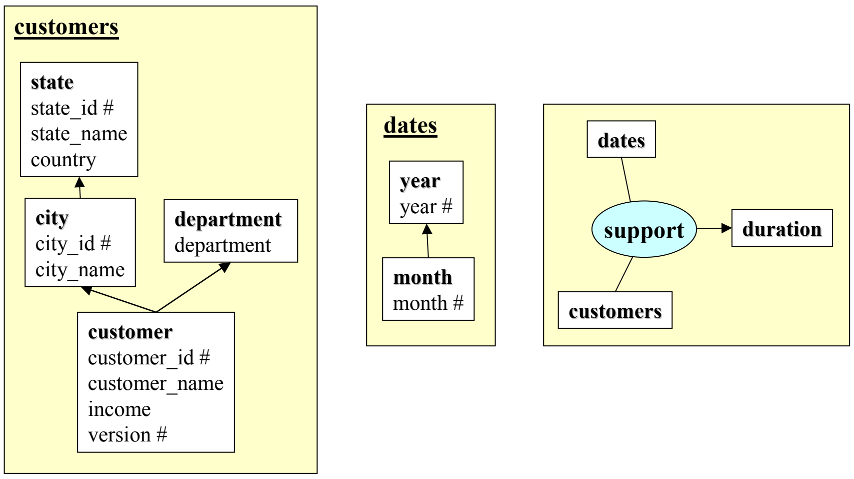


Figure 2. Conceptual schema. Dimension representation consists of levels in hierarchies, which are stated as boxes with their names in bold followed by the attributes (called items). The items followed by a sharp (#) identify the level. Arrows between levels represent dimension hierarchies. Facts are represented by ovals linked to the dimensions, which are stated as boxes. Measures are distinguished with arrows.

The source database has three master tables: *customers*, *cities* and *states*, a table with the customers' incomes and a table that registers customer calls (Figure 3):

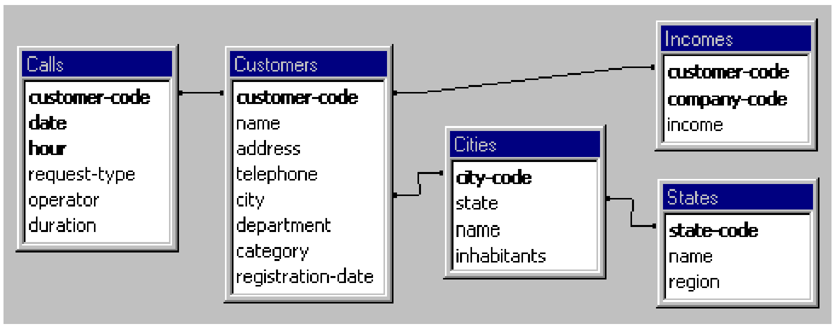


Figure 3. Source database. The boxes represent tables in the relational model. In the top part appears the table name, followed by its attributes. The attributes in bold represent the primary key and the lines between tables (*links*) indicate the attributes used to join them, though complex join conditions are also supported [25].

The logical design process starts from the conceptual schema and the source database. The first task consists in the refinement of the conceptual schema by adding non-functional requirements through *guidelines*.

Guidelines enable to state the desired degrees of normalization [23] or fragmentation [16] in a high level way. Concerning the degree of normalization, there are several alternatives to define dimension tables. For example, for the *customers* dimension, we can store each level in a different table, having attributes for each item of the level and additional items to perform joins (Figure 4a). Other options are to maintain only one table for the whole dimension (Figure 4b), or to follow intermediate strategies. The designer must decide which levels to store together in the same table (by means of the guideline: *Vertical Fragmentation of Dimensions*), based on several criteria like performance, redundancy and disk storage.

Furthermore, *guidelines* enable to define which aggregations (called cubes [8]) are going to be implemented, obtaining derived fact tables. For example, the more detailed cube for the *support* relation is per month and customer (Figure 4c) but the designer may want to store totals, per city and quarter (Figure 4d). The designer must decide which cubes to materialize (*Cube Materialization* guideline), trying to obtain a balance between performance and storage.

We can also indicate the degree of fragmentation of fact tables [24] (Horizontal Fragmentation of Cubes guideline), in order to manage tables with fewer records and improve query performance, for example, storing a table with data of the last two years and another with historical data.

Finally, guidelines can be explicitly defined by the designer or derived using other specific techniques, for example [23] [16].

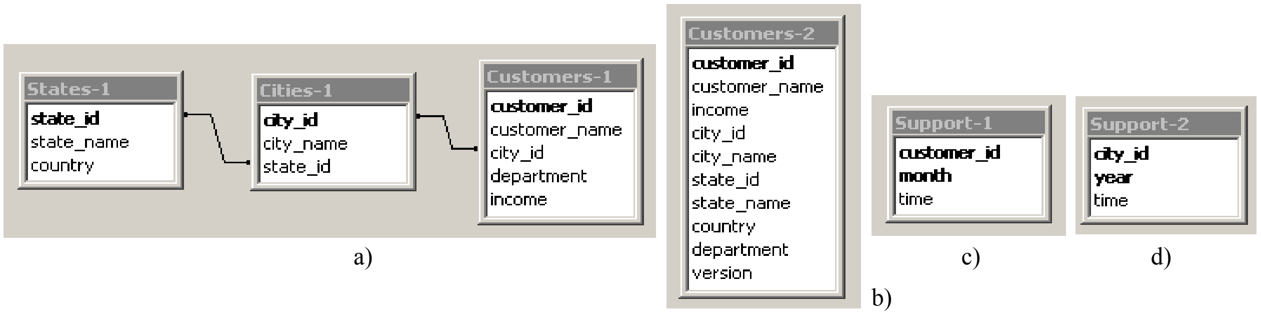


Figure 4. Different strategies to generate relational structures: (a) normalize dimensions, (b) denormalize dimensions, (c) store maximum detail fact tables, and (d) store aggregates.

The second design task is to relate the refined conceptual schema with the source database. For example, we can indicate that the items of the *city* level are calculated respectively from the *city-code* and *city-name* attributes of the *Cities* table and the *month* and *year* items of the *dates* dimension are obtained from the *date* attribute of the *Calls* table performing a calculation. This is done through the *schema mappings.*

Finally, the DW schema has to be generated. In order to do this manually, the designer takes a sequence of decisions that transform the resulting DW schema in each step. Different transformations can be applied depending on the different input information and the design strategies.

For example, if we want to denormalize the *customers* dimension, we have to store attributes corresponding to all the dimension items in the dimension table. This implies joining *Customers, Cities* and *States* tables and projecting the desired attributes. But if we want to normalize the dimension this transformation is not needed. Furthermore, if an item maps to a complex calculated expression several transformations are needed to calculate an attribute for it, for example, for calculating the *income* item as the sum of all customer job's incomes, stored in the *Incomes* table (sum(Incomes.income)). Analogous decisions can be taken for selecting which aggregates to materialize, fragmenting fact tables, filtering data, versioning or removing additional attributes.

We propose to automate most of this last task by using a set of rules. The rules check the conditions that hold and determine which transformations are to be applied. The rule conditions consider the conceptual schema, the source database, mappings between them, and additional design guidelines. The application of the rules follows the design strategies expressed in the guidelines, and is guided by the existence of complex mapping expressions.

4 Guidelines and Mappings

In this section we formalize the concepts of guidelines and mappings introduced in the previous example.

4.1 Design guidelines

Through the guidelines, the designer defines the design style for the DW schema (snowflakes, stars, or intermediate strategies [23]) and indicates performance and storage requirements.

This work proposes guidelines related to the above-mentioned *Cube Materialization*, *Horizontal Fragmentation of Cubes* and *Vertical Fragmentation of Dimensions*.

Cube 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 Figure 2 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 [21]).

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).

- $\text{SCHCUBES} \subseteq \{ <\text{CNAME, R, LS, M}> /$
  $\quad \text{CNAME} \in \text{STRINGS} \wedge$
  $\quad \text{R} \in \text{SCHFACTS} \wedge$
  $\quad \text{LS} \subseteq \{ \text{L} \in \text{GETLEVELS(D)} \ / \ \text{D} \in \text{GETDIMENSIONS(R)} \} \wedge$
  $\quad \text{M} \in (\text{LS} \cup \perp)$
  $\}$[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 5 shows the graphical notation for cubes. There are two cubes: detail and summary of the *support* fact of Figure 2. Their levels are: *month, customer and duration,* and *year*, *city and duration*, respectively. Both of them have duration as measure.

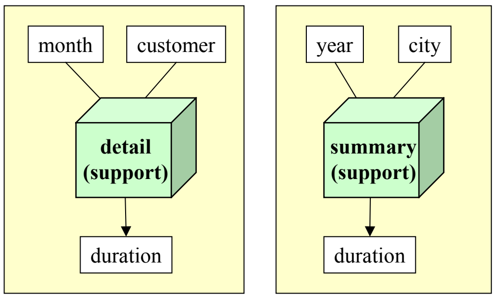


Figure 5. Cubes. They are represented by cubes, linked to several levels (text boxes) that indicate the degree of detail. An optional arrow indicates the measure. Inside the cube are its name and the fact name (between brackets).

Horizontal fragmentation of cubes

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 5 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

[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.

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 fragments, and it is *disjoint* if each tuple belongs to only one fragment. As 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 5. 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 6 shows the graphical notation for strips. There are two strips for the *detail* cube: *actual* and *history*, for calls posteriors and previous to 2002, respectively.

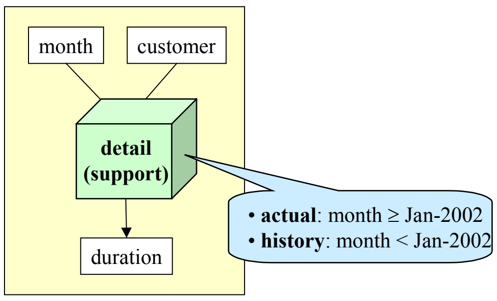


Figure 6 – Strips. The strip set for a cube is represented by a callout containing the strip names followed by their predicates.

[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.

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 fragment is valid only if all its levels are hierarchically related. For example, consider a fragment of the *customers* dimension of Figure 2 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.

$$
\begin{array}{l}
\blacksquare\ \text{SchFragments} \subseteq \{ <\text{FName}, D, \text{Ls}> / \\
\qquad \text{FName} \in \text{Strings} \wedge \\
\qquad D \in \text{SchDimensions} \wedge \\
\qquad \text{Ls} \subseteq \text{GetLevels}(D) \wedge \\
\qquad \forall A,B \in \text{Ls} \,.\, (\\
\qquad\quad <A,B> \in \text{GetHierarchies}(D) \vee \\
\qquad\quad <B,A> \in \text{GetHierarchies}(D) \vee \\
\qquad\quad \exists\, C \in \text{Ls} \,.\, (<A,C> \in \text{GetHierarchies}(D) \wedge <B,C> \in \text{GetHierarchies}(D))\ \ ) \\
\quad \}
\end{array}
$$
[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 7 shows three alternatives to fragment the *customers* dimension.

[3] SchDimensions is the set of dimensions of the conceptual schema. The function GetHierarchies returns the pairs of levels that conform the dimension hierarchies [8].

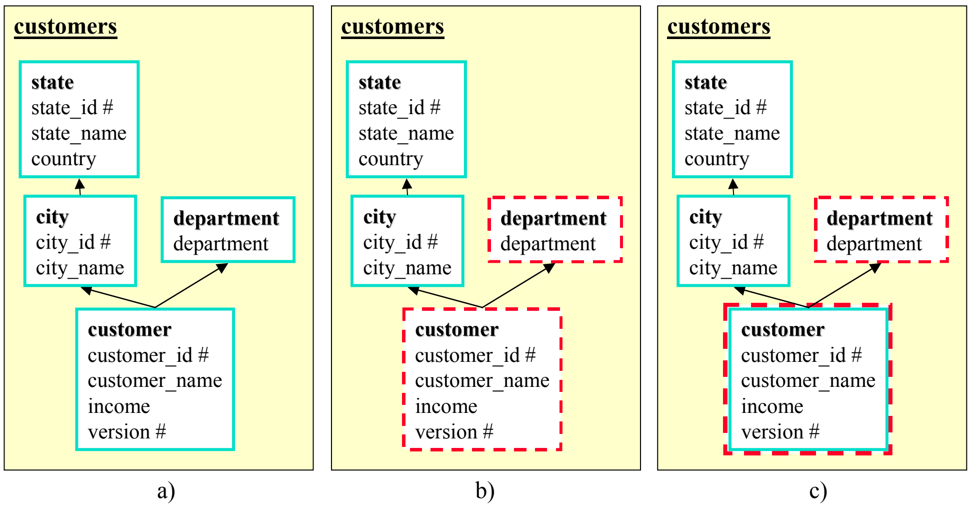


Figure 7. Alternative fragmentations of *customers* dimension. The first one has only one fragment with all the levels. The second alternative has two fragments, one including *state* and *city* levels (continuous line), and the other including the *customer* and *department* levels (dotted line). The last alternative keeps the *customer* level in two fragments (dotted and continuous line).

4.2 Criteria for the definition of guidelines

The conceptual schema is refined with:

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

To specify the guidelines, the designer should consider performance and storage constraints.

For each fact, a set of cubes is specified. It is reasonable to materialize a maximum detail cube to do not loose information. If there are hard storage constraints, only one cube for each fact can be materialized. If not, several cubes can be materialized in order to improve performance for the most frequent or complex queries. As it is not feasible to materialize all possible cubes, the decision is a compromise between storage and performance.

The way of fragmenting the dimensions is related to the design style. To achieve a star schema we define only one fragment for each dimension (with all the levels). However, to define a snowflake schema we define a fragment for each level. Denormalization achieves better query response time but has the maximum of 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, table size grows quickly and performance degrades. Normalization has the opposite effects. Intermediate strategies try to find a balance between query response time and redundancy.

To horizontally fragment a cube we have to study two factors: the tuple number and the subset of tuples that are frequently queried together. If we have more strips, each one has a lower size and we have better response time for querying it. On the other hand, if some queries involve several strips, operations between several tables are needed, slowing the response time. The decision must be based on the requirements, looking at which data is queried together. The most frequents fragmentations take into account ranges of dates or geographical regions.

Default strategies that automate the guideline definition can be studied, for example building only star schemas at maximum detail cube and no strips. Better strategies should study cost functions and heuristics, taking into account storage constraints, performance, affinity and data size. Some ideas are presented in [15].

4.3 Mappings between the refined conceptual schema and the source database

Correspondences or mappings between different schemas are widely used in different contexts [29][12][30][31]. Most of them consider only equivalence correspondences, but in the context of DW design, other types of correspondences are needed to represent calculations and functions over the source database structures. In our proposal, the mappings give the correspondence between the different conceptual schema objects and the logical schema of the source database. They are functions that associate each conceptual schema item with a mapping expression.

Mapping expressions.

A mapping expression is an expression that is built using attributes of source tables. It can be: (i) an *attribute* of a source table, (ii) a *calculation* that involve several attributes from a tuple, (iii) an *aggregation* that involves several attributes of several tuples, or (iv) an *external* value like a constant, a time stamp or version digits. As examples of expressions we have: Customers.city, Year (Customers.registration-date), Sum (Incomes.income), and "Uruguay", respectively.

The specification of the mapping expressions is given in Definition 4. *AttributeExpr* is the set of all the attributes of source tables. *CalculationExpr* and *AggregationExpr* are built from these attributes, combining them through a set of predefined operators and roll-up functions. (The *Expressions* and *RollUpExpressions* functions recursively define the valid expressions. They are specified in [25].) *ExternalExpr* is the union of constants (set of expressions defined from an empty set of attributes), time stamps and version digits. The latter are defined through the *TimeStamp* and *VersionDigits* functions.

- MAPEXPRESSIONS ≡ ATTRIBUTEEXPR ∪ CALCULATIONEXPR ∪ AGGREGATIONEXPR ∪ EXTERNALEXPR
- ATTRIBUTEEXPR ≡ { E ∈ ATTRIBUTES(T) / T ∈ SCHTABLES }
- CALCULATIONEXPR ≡ { E ∈ EXPRESSIONS(AS) / AS ⊆ ATTRIBUTEEXPR }
- AGGREGATEEXPR ≡ { E ∈ ROLLUPEXPRESSIONS(AS) / AS ⊆ ATTRIBUTEEXPR }
- EXTERNALEXPR ≡ EXPRESSIONS(∅) ∪ {TIMESTAMP(), VERSIONDIGITS() }

Definition 4 – Mapping expressions.

Mapping functions.

Given a set of items (Its), a mapping function associates a mapping expression to each item of the set. We also define the macro Mapping(Its) as the set of all possible mapping functions for a given set of items.

- MAPPINGS(ITS) ≡ { F / F : ITS → MAPEXPRESSIONS }

Definition 5 – Mapping functions.

Mapping functions are used in two contexts: in order to map dimension fragments with source tables (fragment mappings), and to map cubes with source tables (cube mappings). In both contexts, each fragment or cube item is associated with a mapping expression.

Consider, for example, a fragment of the *customers* dimension, which includes *customer* and *department* levels. We define a mapping function for the fragment: F, as follows:

- F (customer_id) = Customers.customer-code
- F (customer_name) = Customers.name
- F (income) = SUM (Incomes.income)
- F (version) = VersionDigits()
- F (department) = Customers.department
- F (city_id) = Customers.city

The mapping function maps the *version* item to an *external* expression (version digits), the *income* item to an *aggregation* expression obtained from the *income* attribute of the *Incomes* table, and the other items to *attribute* expressions. The *city_id* item, which is a key of a higher level, is also mapped to allow posterior joins between the generated dimension tables. Figure 8 shows the graphical representation of the mapping function.

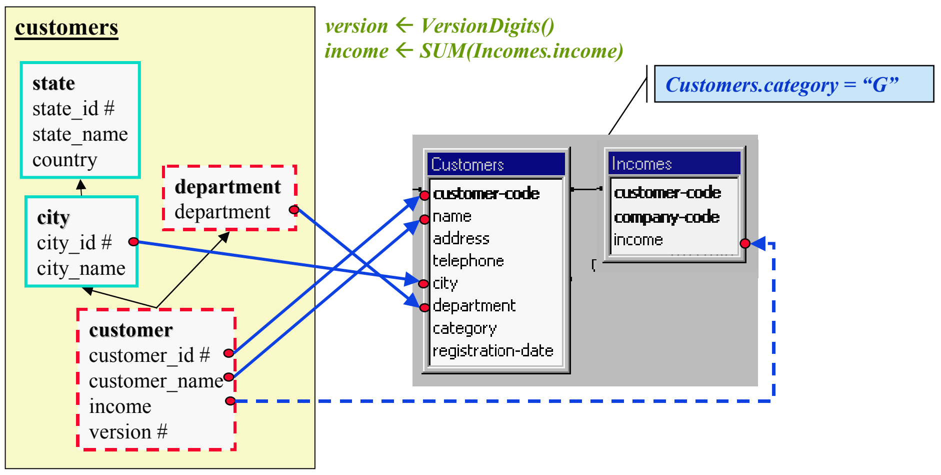


Figure 8. Mapping function. Arrows represent a mapping function from the conceptual schema items to attributes of the source tables. For *attribute* expressions continuous lines are used, for *calculation* and *aggregation* expressions dotted lines linking the attributes involved in the expression are used, enclosing the calculations involved, and for *external* expressions, no lines are used but the calculations are enclosed.

Source database metadata, such as primary keys, foreign keys or any other user-condition involving attributes of two tables (called links), are used to relate the tables referenced in a mapping function.

Further conditions can be applied while defining a mapping function for a fragment or cube, e.g. constrain an attribute to a certain range of values. In the previous example, a condition has been defined: the *category* attribute of *Customers* table must have the value "G". Graphically, conditions are represented as callout boxes as can be seen on Figure 8. Formally, conditions are predicates involving attributes of source tables [25].

For mapping a fragment the designer must define a mapping function and optional conditions. For mapping a cube in addition to the mapping function, he must define other elements, like measure roll-up operators. These definitions can be consulted in [25].

Finally, we say that a mapping function *references* a table if it maps an item to an expression that includes an attribute of the table, and we say that it *directly references* a table if the expression is either an attribute or a calculation (aggregations are not considered). If the mapping function directly references only one table, we say that the table is its *skeleton*.

5 The rule-based mechanism

Different DW design methodologies [16][21][3][23][15][2][19] divide the global design problem in smaller design sub-problems and propose algorithms (or simply sequences of steps) to order and solve them.

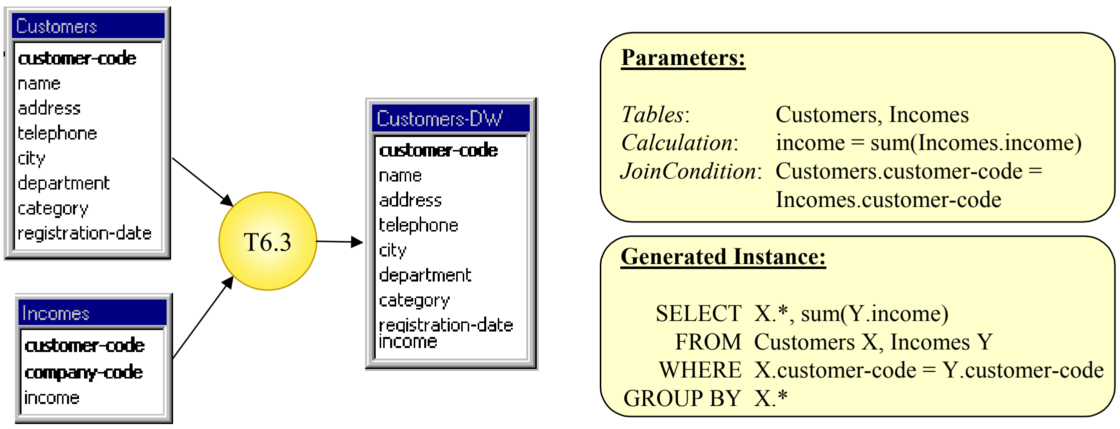
The here proposed mechanism makes use of rules to solve those concrete smaller problems (e.g. related with key generalization). At each design step different rules are applied.

The proposed rules intend to automate the sequence of design decisions that trigger the application of schema transformations and consequently the generation of the DW schema. Alike the designer behaviour rules take into account the conceptual and source schemas as well as the design guidelines and mappings, and then perform a schema design operation.

5.1 The schema transformations

In order to generate the DW schema, the design rules invoke predefined schema transformations which are successively applied to the source schema. We use the set of predefined transformations presented in [22]. Each one takes a sub-schema as input and generates another sub-schema as output, as well as an outline of the transformation of the corresponding instance. These schema transformations perform operations such as: table fragmentation, table merge, calculation, or aggregation.

Figure 9 shows an example, where the transformation T6.3 (DD-Adding N-N) is applied to a sub-schema. It adds an attribute to a relation, which is derived from an attribute of another relation.



Parameters:

Tables:	Customers, Incomes
Calculation:	income = sum(Incomes.income)
JoinCondition:	Customers.customer-code = Incomes.customer-code

Generated Instance:

```
  SELECT  X.*, sum(Y.income)
    FROM  Customers X, Incomes Y
   WHERE  X.customer-code = Y.customer-code
GROUP BY  X.*
```

Figure 9. Application of the transformation DD-Adding N-N. It is applied to *Customers* and *Incomes* tables and the table *Customers-DW* is obtained. The attribute *income* has been added to the *Customers-DW* table containing the sum (aggregation) of all customer jobs' income. The foreign key has been used to join both tables.

5.2 The rules

Table 1 shows a table with the proposed set of rules. The rules marked with * represent families of rules, oriented to solve the same type of problem with different design strategies.

	Rule	Description	Application Conditions
R1	Merge	Combines two tables generating	The mapping function of a fragment or cube directly

		a new one. It is used to denormalize.	references two tables (possibly more). In source metadata, there is a join condition defined between both tables. [4]
R2	Higher Level Names	Renames attributes, using the names of an item that maps to it.	An item of a fragment or cube maps to an attribute, but its name is different to the attribute name.
R3	Materialize Calculation *	Adds an attribute to a table to materialize a calculation or an aggregation.	An item of a fragment or cube maps to a calculation or aggregation expression. The mapping function directly references only the skeleton.
R4	Materialize Extern *	Adds an attribute to a table to materialize an external expression.	An item of a fragment or cube maps to an external expression. The mapping function directly references only the skeleton.
R5	Eliminate Non Used Data	Deletes not-mapped attributes, grouping by the other attributes and applying a roll-up function to measures.	A table has attributes that are not mapped by any item. The mapping function directly references only the skeleton.
R6	Roll-Up *	Reduces the level of detail of a fact table, performing a roll-up of a dimension.	A cube is expressed as an aggregate of another one. The mapping function of the second cube directly references only the skeleton.
R7	Update Key	Changes a table primary key, so as to agree the key of a fragment or cube with the attributes that it maps.	The key of a fragment or cube does not agree with the key of the table referenced by the mapping function. The mapping function directly references only the skeleton.
R8	Data Constraint	Deletes from a table the tuples that don't verify a condition.	A mapping has constraints or a horizontal fragmentation is defined. The mapping function directly references only the skeleton.

Table 1. Design rules.

Figure 10 sketches the context of a rule.

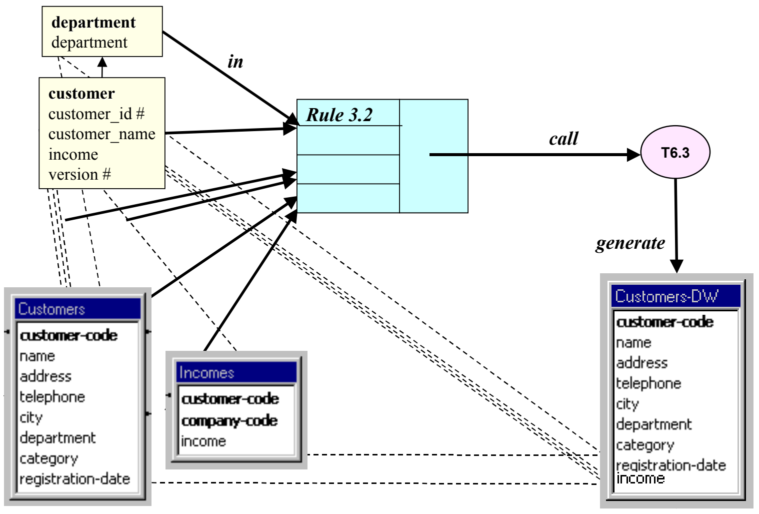


Figure 10. Context of application of a rule. Refined conceptual schema structures, source tables and mappings are the input of the rule. If conditions are verified, the rule calls a transformation, which generates a new table transforming the input ones. Mappings to the conceptual schema are updated, and a transformation trace from the source schema is kept.

[4] The rule is applied successively to pairs of directly referenced tables until the mapping function directly references only one table, called its *skeleton*.

In the next section we show the specification of the rule R3.2 (MaterializeAggregate) of the family R3. The goal of this rule is to materialize an aggregation. The conditions to apply the rule are: (i) the mapping function of a cube or fragment maps an item to an *aggregate* expression, and (ii) the mapping function directly references only the skeleton. The effect of this last condition is to simplify the application of the corresponding transformation, enforcing to apply join operations before (The previous application of the *merge* rule assures that this condition can be satisfied). The rule invokes the transformation T6.3.

Figure 11a shows an application example of Rule R3.2 to a fragment of the *customers* dimension. The *income* item maps to the sum of the attribute *income* of source table *Incomes.* The rule evaluates the conditions and applies the schema transformation DD-Adding N-N, which builds a new table with a calculated attribute (as *SUM(Incomes.income)).*

The rule application also updates the mapping function to reference the new table. Figure 11b shows the updated mapping function.

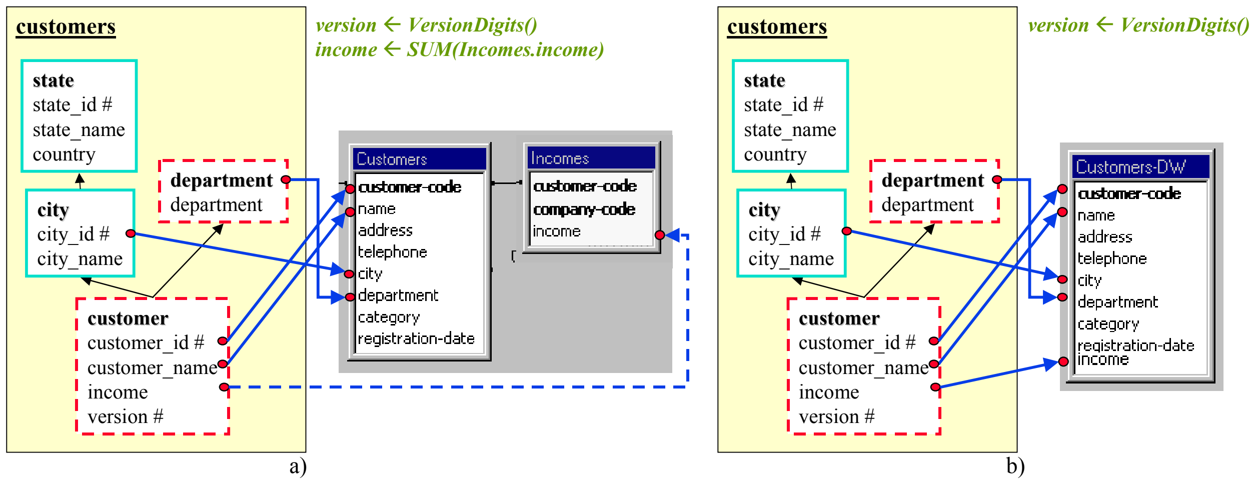


Figure 11. Mapping function of a fragment of *customers* dimension: a) before and, b) after applying the rule

5.3 Rule specification.

Rules are specified in 5 sections: *Description, Target Structures, Input State, Conditions* and *Output State*. The *Description* section is a natural language description of the rule behaviour. The *Target Structures* section enumerates the refined conceptual schema structures that are input to the rule. The *Input State* section enumerates the rule input state consisting of tables and mapping functions. The *Conditions* section is a set of predicates that must be fulfilled before applying the rule. Finally, the *Output State* section is the rule output state, consisting of the tables obtained by applying transformations to the input tables, and mapping functions updated to reflect the transformation. The following scheme sketches a rule:

$$\textbf{RuleName:}\quad \text{Target Structures}\left|\,\frac{\text{Input State} \langle\text{mapping, tables}\rangle}{\text{Output State} \langle\text{mapping, tables}\rangle}\right.\ \text{conditions}$$

Table 2 shows the specification of the rule R3.2 of family R3.

RULE R3.2 – AGGREGATE CALCULATE

Description:

Given an object from the refined conceptual schema, one of its items, its mapping function (that maps the item to an aggregate expression) and two tables (referenced in the mapping function), the rule generates a new table adding the calculated attribute.

Target Structures:

- $S \in$ SchFragments $\cup$ SchCubes *// a conceptual structure: either fragment or cube*
- $I \in$ GetItems(S) *// an item of S*

Input State:

- **Mappings:** $F \in$ Mappings(GetItems(S)) *// the mapping function for S*
- **Tables:** $T_1 \in$ ReferencedTables (GetAttributeExprs(F) $\cup$ GetCalculationExprs(F))
  // a table referenced in all attribute and calculation expressions
  $T_2 \in$ ReferencedTables (F(I)) *// a table referenced in the mapping expressions of the item*

Conditions:

- $F(I) \in$ GetAggregationExprs(F) *// the expression for the item is an aggregation*
- #ReferencedTables (GetAttributeExprs(F) $\cup$ GetCalculationExprs(F)) = 1
  // all attribute and calculation expressions reference to one table

Output State:

- **Tables:**
  - T' = Transformation T6.3 ($\{T_1, T_2\}$, I = F(I), GetLink (T_1, T_2))
    // arguments are the source tables, the calculation function for the item and the join function between the tables.
  - UpdateTable (T', T) *// Update table metadata*
- **Mappings:**
  - $F' \,/\, F'(I) = T'.I \wedge \forall\, J \in \text{GetItems}(S), J \neq I \,\mathbf{.}\, (\, F'(J) = \text{UpdateReference}(F,T,T')\,)$
    // The mapping function will correspond the given item with an attribute expression. The mapping function is updated to reference the new table. [5]

Table 2. Specification of a rule. As Target structures we have a fragment or cube and one of its items. As Input we have the mapping function of the structure and two tables referenced in the mapping function. The first table is referenced by *attribute and calculation* expressions and the second one is referenced by the aggregation expression. The Conditions have been explained previously. The Output table is the result of applying the transformation T6.3 and the Output mapping function is the result of updating mapping expressions in order to reference the new table. The function corresponds the target item with the new attribute.

5.4 Rule execution.

The proposed rules do not include a fixed execution order. Therefore, they need an algorithm that order their application. Our approach is similar to the one followed to perform View Integration in [29]. The algorithm states in which order will be solved the different design problems.

We propose an algorithm that is based on existing methodologies and some practical experiences. The algorithm consists of 15 steps. The first 6 steps build dimension tables. Then, steps 7 to 12 build fact tables, steps 13 and 14 perform the aggregates and finally step 15 carries on horizontal fragmentations. Each step is associated to the application of one or more rules.

We present the global structure of the algorithm and the details of one of its steps (Table 5). The complete specification can be found in [25].

[5] The GETATTRIBUTEEXPRS, GETCALCULATIONEXPRS, GETAGGREGATIONEXPRS and GETEXTERNALEXPR functions return all the attribute, calculation, aggregation and external expressions respectively, that are mapped by the mapping function. The GETREFERENCEDTABLES function returns the tables referenced in a set of mapping expressions. The GETLINK function returns the link predicate defined in the metadata to join the tables.

Algorithm: DW relational schema generation

Phase 1: Build dimension tables for each fragment
- Step 1: Denormalize tables referenced in mapping functions (R1)
- Step 2: Rename attributes referenced in mapping functions (R2)
- Step 3: Materialize calculations (R3, R4)
- Step 4: Data Filter following mapping function conditions (R8)
- Step 5: Delete attributes non-referenced in mapping functions (R5)
- Step 6: Update primary keys of tables referenced in mapping functions (R7)

Phase 2: Build fact tables for each cube
- Step 7: Denormalize tables referenced in mapping functions (R1)
- Step 8: Rename attributes referenced in mapping functions (R2)
- Step 9: Materialize calculations (R3, R4)
- Step 10: Data Filter following mapping function conditions (R8)
- Step 11: Delete attributes non-referenced in mapping functions and apply measure roll-up (R5)
- Step 12: Update primary keys of tables referenced in mapping functions (R7)

Phase 3: Build aggregates
- Step 13: Build an auxiliary table with roll-up attributes (if it does not exist) (R1, R2, R3, R5, R6)
- Step 14: Build aggregates tables (R5, R6)

Phase 4: Build horizontal fragmentations
- Step 15: Data Filter following fragmentations conditions (R8)

STEP 3: MATERIALIZE CALCULATIONS

For each fragment of the refined conceptual schema, apply: rule R3.1 to each calculation expression, rule R3.2 to each aggregation expression, and rule R4 to each external expression.

```
For each S ∈ Fragments  // for each fragment S
    F = FragmentMapping(S)  // F is the mapping function of the fragment
    {T} = ReferencedTables (GetAttributeExprs(F) ∪ GetCalculationExprs(F))
                // T is the table referenced by attribute and calculation expressions. There is only a referenced table
                because of previous execution of step 1
    For each I ∈ GetItems(S)  // for each item of fragment S
        If F(I) ∈ GetCalculationExprs(F)  // F corresponds I to a calculation expression
                Apply Rule3.1:     $S, I \left| \frac{\langle F,T \rangle}{\langle F',T' \rangle} \right.$
        If F(I) ∈ GetAggregationExprs(F)  // F corresponds I to an aggregation expression
                Apply Rule3.2:     $S, I \left| \frac{\langle F,T \rangle}{\langle F',T' \rangle} \right.$
        If F(I) ∈ GetExternalExprs(F)  // F corresponds I to an external expression
                Apply Rule4:       $S, I \left| \frac{\langle F,T \rangle}{\langle F',T' \rangle} \right.$
        T = T'
        F = F'  // update loop variables
    Next
    FragmentMapping(S) = F  // update the mapping function for the fragment
Next
```

Table 3. Specification of a step of the algorithm

5.5 An example of application

In this section we show the application of the rules to the example of section 3.2.

The designer defines guidelines to denormalize the *dates* dimension and have two fragments for the *customers* dimension: *geography* (city, state) and *departments* (customer, department). He also wants to have two cubes: *detail* (by month and customer) and *summary* (by year and city). The first cube will be fragmented with calls after year 2000 and historical data.

The mapping function of the *department* fragment is shown in Figure 8 and the other mapping functions in Figure 12. The *summary* cube will be calculated as a roll-up of the *detail* cube and then no mapping is needed.

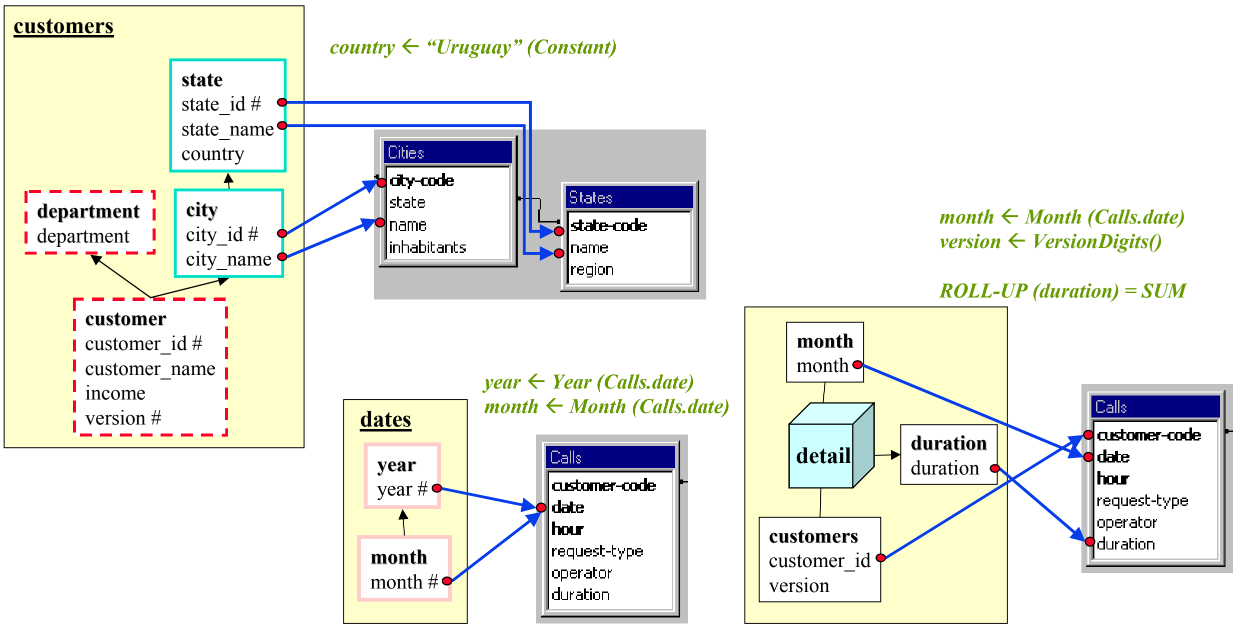


Figure 12. Mapping functions for the example.

The first step consists in denormalizing the fragments. The only fragment that directly references several tables is *geography*. (aggregate expressions are processed later). Rule R1 is executed to merge tables *Cities* and *States*. The rule gives the following table as result:

```
DwGeography01 (city-code, state, name, inhabitants, state-code, name#2, region)
```

Step 2 renames direct mapped attributes. Rule R2 gives the following tables as result of each application:

```
DwGeography02 (city-id, state, city-name, inhabitants, state-id, state-name,
               region)
DwDepartments01 (customer-id, customer-name, address, telephone, city-id,
                 department, category, registration-date)
```

Step 3 materialize calculated attributes. The are two items that map to *calculation* expressions: *month* and *year*, an item that maps to an *aggregation* expression: *income*, and two items that map to extern expressions: *country* and *version*. Rules of family R3 and R4 are applied (as specified in the previous section). The rules give the following tables as result of each application:

```
DwGeography03 (city-id, state, city-name, inhabitants, state-id, state-name, region,
               country)
DwDepartments02 (customer-id, customer-name, address, telephone, city-id, department,
                 category, registration-date, income)
DwDepartments03 (customer-id, customer-name, address, telephone, city-id, department,
                 category, registration-date, income, version)
DwDates01 (customer-code, date, hour, request-type, operator, duration, month)
DwDates02 (customer-code, date, hour, request-type, operator, duration, month, year)
```

Step 4 filters data following mapping constraints. The only fragment with mapping constraints is *department*. As rule R8 is a data operation, the table schema is not changed.

```
DwGeography04 (city-id, state, city-name, inhabitants, state-id, state-name, region,
               country)
```

Step 5 deletes not-mapped attributes and groups by the remaining attributes (then keys are altered). Rule R5 is applied. We obtain the following tables as result of each application.

```
DwGeography05 (city-id, city-name, state-id, state-name, country)
DwDepartments04 (customer-id, customer-name, city-id, department, income, version)
DwDates03 (month, year)
```

Finally, step 6 set primary keys applying rule R7. Conceptual schema key metadata is used by the rule. We obtain the following tables as result of each application.

```
DwGeography06 (city-id, city-name, state-id, state-name, country)
DwDepartments05 (customer-id, customer-name, city-id, department, income, version)
DwDates04 (month, year)
```

Analogously, steps 7 to 12 are applied for the *support* cube. An important difference is that rule R5 also applies roll-up functions to the measures. We obtain the following tables as result of each application.

```
DwDetail01 (customer-id, date, hour, request-type, operator, duration)
DwDetail02 (customer-id, date, hour, request-type, operator, duration, month)
DwDetail03 (customer-id, date, hour, request-type, operator, duration, month, version)
DwDetail04 (customer-id, duration, month, version)
DwDetail05 (customer-id, duration, month, version)
```

Step 13 is executed when the attributes necessaries to make a roll-up (those that correspond to the items that identify the involved levels) are not contained in a same table. This is not the case here, because *month* and *year* (necessaries to make the roll-up by the *dates* dimension) are both contained in the *DwDates04* table, and *customer_id, version* and *city_id* (necessaries to make the roll-up by the *customers* dimension) are contained in the *DwDepartments05* table.

Step 14 build an aggregate fact table for the *summary* cube applying rule R6. We obtain the following tables as result of each roll-up.

```
DwSummary01 (customer-id, duration, version, year)
DwSummary02 (duration, year, city-id)
```

Finally, step 15 fragments the table referenced by the *support* cube, applying rule R8 twice. We obtain the following tables with identical schema.

```
DwSupportActual01 (customer-id, duration, month, version)
DwSupportHistory01 (customer-id, duration, month, version)
```

The resulting DW has the following tables:

```
DwGeography06 (city-id, city-name, state-id, state-name, country)
DwDepartments05 (customer-id, customer-name, city-id, department, income, version)
DwDates04 (month, year)
DwSummary02 (duration, year, city-id)
DwSupportActual01 (customer-id, duration, month, version)
DwSupportHistory01 (customer-id, duration, month, version)
```

6 Conclusions

This paper presents a rule-based mechanism to automate the construction of DW relational schemas. The kernel of the mechanism consists of a set of design rules that decide the application of the suitable transformations in order to solve different design problems. The overall framework integrates relevant elements in DW design: mappings between source and DW conceptual schemas, design guidelines that refine the conceptual schema, schema transformations which generate the target DW schema, and design rules that embed design strategies and call the schema transformations.

The proposed framework is a step forward to the automation of DW logical design. It provides an open and extensible environment that enables to apply existing design techniques into a unique place allowing for enhanced productivity and simplicity. Moreover, new design techniques can be integrated as new rules or transformations.

This framework has been prototyped [28][25][13]. 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.

We are currently working in the extension of the framework to support multiple source databases. We also implement a CWM based repository to be used by the global CASE environment.

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 using the information provided by the mappings and the transformation trace. Particular effort is been taken in adapting our system to automatically suggest the design guidelines – based on heuristic cost functions dependent on physical storage constraints, query performance, affinity or amount of data.

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] Bouzeghoub, M.; Fabret, F.; Matulovic-Broqué, M.: “*Modeling Data Warehouse Refreshment Process as a Workflow Application*”. DMDW’99, Germany, 1999.

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

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

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

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

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

[12] Fankhauser, P.: “A Methodology for Knowledge-Based Schema Integration”. PhD-Thesis, Technical University of Vienna, 1997. ?

[13] Garbusi, P.; Piedrabuena, F.; Vázquez, G.: “Diseño e Implementación de una Herramienta de ayuda en el Diseño de un Data Warehouse Relacional”. Undergraduate project. Universidad de la República, Uruguay, 2000.

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

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

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

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

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

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

[20] Kenan Technologies:"An Introduction to Multidimensional Databases". White Paper, Kenan Technologies, 1996.

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

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

[25] Peralta, V.: "Diseño lógico de Data Warehouses a partir de Esquemas Conceptuales Multidimensionales". Master Thesis. Universidad de la República, Uruguay. 2001.

[26] Peralta, V. Martota, A.: "Hacia la Automatización del Diseño de Data Warehouses". CLEI'2002. Uruguay. 2002.

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

[28] Proyectos de Ingeniería de Software 2001. "*Implementación de herramientas CASE que asistan en el Diseño de Data Warehouses*". Software engineering thesis. Universidad de la República, Uruguay, 2001.

[29] Spaccapietra, S. Parent, C.: "View integration: A step forward in solving structural conflicts". TKDE'94, Vol 6, No. 2, 1994.

[30] Vidal, V. Lóscio, B. Salgado, A.: "Using correspondence assertions for specifying the semantics of XML-based mediators". WIIW'2001, Brazil, 2001.

[31] Yang, L. Miller, R. Haas, L. Fagin, R.: "Data-Driven Understanding and Refinement of Schema Mappings". ACM SIGMOD'01, USA, 2001.