Designing Data Warehouses

through schema transformation primitives

Adriana Marotta, Raul Ruggia
Computer Science Department – University of the Republic of Uruguay
adriana@fing.edu.uy

Abstract

A Data Warehouse (DW for short) is a Database that stores information in order to satisfy decision making requests. The features of DWs cause that the design process and the used strategies be different from the traditional ones for Relational Databases. This paper addresses the DW design problem. Its approach is based in schema transformations that are applied to a source schema in order to generate a DW schema. We define a set of DW design primitives. These primitives materialize design criteria knowledge and provide a way for tracing the design. In addition, they behave as DW design building blocks that can be composed for building the final schema. We also present a set of consistency rules that must be applied to ensure consistency of the obtained result, and some strategies that provide different solutions to typical problems that must be faced during the design of a DW.

1. Introduction

A Data Warehouse (DW for short) is a Database that stores information in order to satisfy decision-making requests. This kind of Database has the following particular features. It contains data that is the result of transformations, quality improvement, and integration of data that comes from operational bases, also including indicators that gives it additional value. The DWs have to support complex queries (summarisation, aggregates, crossing of data), although its maintenance does not suppose transactional load.

The features of DW cause that the design process and the used strategies to be different from the traditional ones for Relational Databases. For example, in DW design, the existence of redundancy in data is necessary for improving performance of complex queries and it does not imply problems like data update anomalies. This is because, in general, DWs' maintenance is performed by means of controlled batch loads.

This paper addresses the DW Design problem through a schema transformation approach.

The main interest for the definition of design primitives is twofold: first, primitives materialize design criteria knowledge, second, they provide a way for tracing the design. In addition, they increase designer's productivity by behaving as design building blocks that can be composed for building the final schema. This work can be considered as an important step toward the introduction of DW design features into a CASE Tool.

The use of schema transformation primitives is a classical conceptual tool in Databases area. In [1], Batini, Ceri and Navathe present design primitives and strategies as the building blocks of conceptual design methodologies. In [2], Hainaut analyses the concept of schema transformation and generalises many of the proposed transformations in a conceptual schema design context. In [3], database schema transformations are used and automated to perform schema evolution and reorganization. This work proposes a set of primitives for relational DW design.

The primitives we present have been designed having into account various DW data models and design strategies. Practical design techniques and methods have been proposed by Kimball in [4], [5], and [6], following mainly a star-schema approach. In [7], Adamson and Venerable also present concrete solutions for different target business. In [8], Silverston, Inmon and Graziano present DW models in a pattern-oriented approach, and propose techniques for converting a corporate logical data model into the DW model.

In general, existing work in DW design consists mainly of techniques for specific sub-models (as star or snowflake) and design patterns for specific domain areas. Although this work constitutes a precious knowledge base in DW design its practical application is not direct. In order to do it, designers must incorporate this knowledge, abstract the design rules and strategies, and then apply them in particular cases. Furthermore, this application would not be structured in well-defined design steps.

The present work intends to abstract and structure DW design techniques and strategies in a set of schema transformation primitives. In addition includes some guidelines for their application.

The main contribution of this work is the proposal of a set of DW schema design primitives. These primitives are to be applied to the source schemas, more specifically to their integration. Together, with each primitive, this work provides the specification of the transformation that must be applied to the source schema instances in order to populate the generated DW.

For the utilisation of the primitives, we provide two types of guidelines: a set of consistency rules that must be applied always to ensure the consistency of the obtained result, and some strategies that provide different solutions to typical problems that must be faced during the design of a DW.

The remainder of paper is organized as follows. Section 2 presents an overview of the primitives. Section 3 provides basic definitions: the used model and a set of schema invariants. Section 4 shows the primitives implemented. Section 5 is an overview of some rules and strategies for the application of the primitives and Section 6 concludes.

2. The schema transformation primitives – An overview

In our approach, DW design is a process that starts with a source schema and ends with a result schema, which corresponds to the DW schema. This is generated by application of primitives to the source schema and to the intermediate sub-schemas[1] that are generated through the process, that is to say, during the design the primitives are composed to obtain the wished final schema. Therefore, all the elements that constitute the final schema, relations and attributes, are the result of applying the primitives to the source schema.

Architecture of the transformation

Figure 2.1 shows the basic architecture of the transformation of a source schema in a DW schema, through the application of primitives.

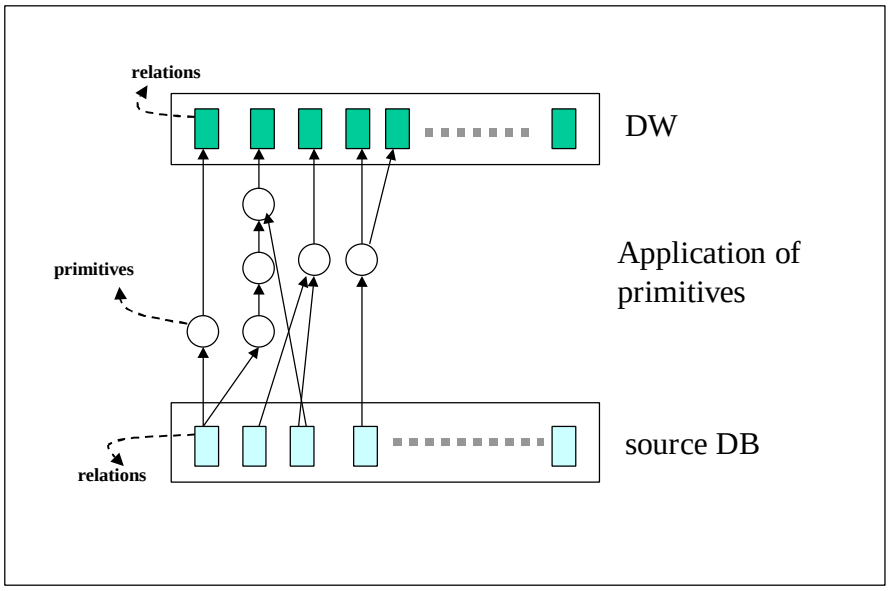


Figure 2.1

The primitives we have designed, are basic transformations that are applied to a Relational database schema in order to obtain a Relational DW schema. Roughly speaking, they take as input a sub-schema and their output is another sub-schema. The transformations to be applied to the source instance, are sketched.

We group some of the primitives into families because in some cases there are several alternatives for solving the same problem, or more than one style of design that can be applied.

[1] In this work, we consider a sub-schema as a set of relations.

The primitives do not lead to some strategy or methodology in particular. Moreover, their application without well-defined design criteria, can lead to undesired results, sometimes having consistency problems. In order to help the designer in this aspect we provide the following: (i) the definition of DW schema invariants (Section 3), (ii) strategies for solving typical problems that appear in DW design (Section 5.1), and (iii) consistency rules for application of primitives (Section 5.2).

3. Basic Definitions

The underlying model for the proposed transformation primitives is the Relational Model. In addition, the relational elements (relations and attributes) are classified into different sets, according to their behaviour in a DW context.

As a glance, some of the classified elements are: dimension relations, measure relations, descriptive attributes, measure attributes.

This classification enables the primitives to perform a more refined treatment of the different situations in DW design.

The Sets defined over the Relational Model

Relation[2] sets:

Rel – Set of all the relations (any kind of relation).

Rel_D – Set of "dimension" relations. These are the relations that represent descriptive information about real world subjects.

Rel_M – Set of "measure" relations. These are the relations that represent relationships or combinations among the elements of a group of dimensions. Usually, they contain attributes that represent measures for the combinations.

Rel_C – Set of "crossing" relations. These are the measure relations that do not have any measure attribute.

Rel_J – Set of "hierarchy" relations. These are the dimension relations that contain a set of attributes that constitute a hierarchy. The fact that exists a hierarchy among a set of attributes, can only be determined having into account the semantic of them.

$Rel_H(R)$ – Set of "history" relations. These are the relations that have historical information that correspond to information in relation R.

These sets verify the following properties:

- $Rel_C \subset Rel_M$

[2] In this work, we use the word relation as a synonym of relation schema.

- $Rel_J \subset Rel_D$
- $\forall$ R $Rel_H(R) \subset (Rel_D \cup Rel_M)$

Attribute sets:

$Att(R)$ – Set of all attributes of relation R.

$Att_M(R)$ – Set of measure attributes of relation R.

$Att_D(R)$ – Set of descriptive attributes of relation R.

$Att_C(R)$ – Set of derived (calculated) attributes of relation R.

Att_J – Set of sets of attributes that represent a hierarchy.

$Att_K(R)$ – Set of sets of attributes that are key in relation R.

$Att_{FK}(R)$ – Set of sets of attributes that are foreign key in relation R.

$Att_{FK}(R_1, R_2)$ – Set of attributes that are foreign key in relation R_1 with respect to relation R_2.

These sets verify the following properties:

- $Att_M(R) \cup Att_D(R) \cup Att_C(R) = Att(R)$
- $\forall$ X / X $\in Att_J$, X $\subset \cup_{R \in Rel} Att_D(R)$
- $Att_{FK}(R)$ = { e / e = $Att_{FK}(R, R_i)$ }, i=1..n, where n is the number of relations with respect to which R has a foreign key.
- $\forall$ A / A $\in$ X and X $\in (Att_K(R) \cup Att_{FK}(R))$, A $\in Att_D(R)$
- If X $\in Att_K(R)$ and Y $\in Att_{FK}(R)$, it may be: X $\cap$ Y $\neq \varnothing$

DW Schema Invariants

Schema invariants are a set of properties that must be satisfied by a relational DW schema for being consistent. The invariants are:

I1 - Referential integrity :

Each declared foreign key must have a corresponding primary key in the relations it references. Besides it must reference to all relations with this primary key.

$\forall$ X, R_1, R_2 / X $\in Att_{FK}(R_1, R_2)$, it holds:

$$X \in Att_K(R_2) \wedge$$
$$\forall R / X \in Att_K(R), \; X \in Att_{FK}(R_1, R)$$

I2 - Hierarchies :

Given a set of attributes X representing a hierarchy, it must hold a functional dependency between each attribute of X and all attributes of X that correspond to higher levels in the hierarchy.

Let $X / X \in Att_J \wedge X = \{A_1, ..., A_n\} \wedge$

$A_1 < A_2 < .... < A_n$, where a<b means that b represents a higher level in the hierarchy than a

it holds $A_1 \rightarrow A_2$

$A_2 \rightarrow A_3$

.....

$A_{n-1} \rightarrow A_n$

I3 - History relations :

- A history relation that corresponds to a relation with current data, must include a foreign key referencing to the corresponding current relation.

  Let $R_H / R_H \in Rel_H(R)$, it holds that $\exists X / X = Att_{FK}(R_H,R)$

I4 - Measure relations :

- If a measure relation has an attribute from some dimension relation, then it must have a foreign key relative to this relation.

  Let $R_D, R_M / R_D \in Rel_D \wedge R_M \in Rel_M$

  if $\exists A / A \in Att(R_D) \wedge A \in Att(R_M) \Rightarrow \exists X / X = Att_{FK}(R_M,R_D)$

- Measure relations must have a functional dependency, whose left-hand side is the set of attributes that are foreign keys to dimensions and right-hand side are the rest of attributes.

  Let $R_M, X / R_M \in Rel_M \wedge X = Att_{FK}(R_M)$, it holds $X \rightarrow (Att(R_M) - X)$

4. The primitives

The primitives we propose are transformation operations that can be applied to a schema to make it more suitable for queries that will be submitted to it.

The following examples illustrate their usefulness.

Many relations in operational systems do not maintain a temporal notion. For example, stock relations use to have the current stock data, updating it with each product movement. However, in DWs most relations need to include a temporal element so that they can maintain historical information. For this purpose, there is a primitive called **Temporalization** that adds an element of time to the set of attributes of a relation.

In production systems, usually, data is calculated from other data at the moment of the queries, in spite of the complexity of some calculation functions, in order to prevent any kind of redundancy. For example, the product prices expressed in dollars are calculated from the product prices expressed in some other currency and a table containing the dollar values. In a DW system, sometimes it is convenient to maintain this kind of data calculated, for performance reasons. We have a group of primitives, whose name is **DD-Adding**, that add to a relation an attribute that is derived from others.

In operational databases information in measure relations are stored at the highest level of detail that is possible. Usually, in these relations exist some attribute that has a hierarchy associated. Often, when this is passed to a DW, it is useful to summarize data by that attribute following the hierarchy (doing a "roll-up"). For example, data about movements in a stock system, that is stored in a daily level, need to be stored by monthly totals in the DW. In this case we have to do a roll-up in the hierarchy of time. A relation can be generated for this purpose, through a primitive called **Hierarchy Roll Up**, which also can generate a new hierarchy relation with the corresponding grain.

Figure 4.1 shows a table containing the whole set of primitives proposed.

Primitive		Description
P1	**Identity**	Given a relation, it generates another that is exactly the same as the source one.
P2	**Data Filter**	Given a source relation, it generates another one where only some attributes are preserved. Its goal is to eliminate purely operational attributes.
P3	**Temporalization**	It adds an element of time to the set of attributes of a relation.
P4	**Key Generalization ***	These primitives generalize the primary key of a dimension relation, so that more than one tuple of each element of the relation can be stored.
P5	**Foreign Key Update**	Through this primitive, a foreign key and its references can be changed in a relation. This is useful when primary keys are modified.
P6	**DD-Adding ***	The primitives of this group add to a relation, an attribute that is derived from others.
P7	**Attribute Adding**	It adds attributes to a dimension relation. It should be useful for maintaining in the same tuple more than one version of an attribute.
P8	**Hierarchy Roll Up**	This primitive does the roll up by one of the attributes of a relation following a hierarchy. Besides, it can generate another hierarchy relation with the corresponding level of detail.
P9	**Aggregate Generation**	Given a measure relation, this primitive generates another measure relation, where data are resumed (or grouped) by a given set of attributes.
P10	**Data Array Creation**	Given a relation that contains a measure attribute and an attribute that represents a pre-determined set of values, this primitive generates a relation with a data array structure.
P11	**Partition by Stability ***	These primitives partition a relation, in order to organize its history data storage. Vertical Partition or Horizontal Partition can be applied, depending on the design criterion used.
P12	**Hierarchy Generation ***	This is a family of primitives that generate hierarchy relations, having as input, relations that include a hierarchy or a part of one.
P13	**Minidimension Break off**	This primitive eliminates a set of attributes from a dimension relation, constructing a new relation with them.
P14	**New Dimension Crossing**	This primitive allows to materialize a dimension data crossing in a new relation. It also is useful to simply de-normalize relations, which improves queries performance.

Figure 4.1

Some specifications of primitives

In this section we show specifications of some of the primitives. Specifications of the whole set of primitives defined can be found in [9].

The following specifications present four sections. The **Description** specifies a common language description about the primitive behaviour. The **Input** specifies the source schema and other arguments. The **Resulting schema** is the specification of the schema that would be generated by the primitive. The **Generated instance** is a sketch of the transformation to be applied to the instance of the source schema.

<u>Primitive P4.2</u> – Key Extension

Description:

Given a dimension relation, the key is generalized. It is extended by the inclusion of new attributes of the relation into it.

Input:

- source schema : $R (A_1, ..., A_n) \in Rel_D \ / \ \exists X \subset \{ A_1, ..., A_n \} \wedge X \in Att_K(R)$
- $Y \subset (\{ A_1, ..., A_n \} - X)$, attributes to be included in the key
- source instance : r

Resulting schema:

$R' (A_1, ..., A_n) \in Rel_D \ / \ \exists Z \subset \{ A_1, ..., A_n \} \ \wedge \ Z \in Att_K(R') \ \wedge \ Z = XY$

Generated instance:

r' = r

Primitive P9 – Aggregate Generation

Description:

Given a measure relation, the primitive generates another measure relation, where data are resumed (or grouped) by a given set of attributes.

Input:

- source schema : $R (A_1, ..., A_n) \in Rel_M$
- Z , set of attributes / card(Z) = k (measures)
- $\{ e_1, ..., e_k \}$, aggregate expressions
- $Y / Y \subset \{ A_1, ..., A_n \} \wedge Y \subset (Att_D(R) \cup Att_M(R))$, attributes to be removed
- source instance : r

Resulting schema:

$R' (A'_1, ..., A'_m) \in Rel_M$ / $\{ A'_1, ..., A'_m \} = \{ A_1, ..., A_n \} - Y \cup Z$

Generated instance:

r' = select ($\{ A'_1, ..., A'_m \} - Z) \cup \{ e_1, ..., e_k \}$
from R
group by $\{ A'_1, ..., A'_m \} - Z$

Example:

We have a relation, with the quantities sold by customer, salesman, month, product and city.

MONTH_SALES

CUSTOMER	SALESMAN	MONTH	PROD	CITY	QUANTITY
Juan	Pedro	1/98	25	Montevideo	5
Juan	Pedro	1/98	7	Colonia	7
Juan	Maria	2/98	4	Montevideo	1
Juan	Laura	2/98	4	Maldonado	5
Luis	Pedro	1/98	100	Montevideo	2
Luis	Laura	1/98	100	Montevideo	6
Luis	Laura	4/98	100	Canelones	3

Now we want to store the quantities that were sold by each customer on each month and of each product. Therefore we will group by CUSTOMER, MONTH, PRODUCT.

We apply primitive **P9**, where the input is:

- R = MONTH_SALES
- Z = { QUANTITY }, card(Z) = k = 1, the measure we want to appear
- { e_1, ..., e_k } = { sum(QUANTITY) }
- Y = { SALESMAN, CITY }
- r = tuples of MONTH_SALES

Result:

CUST_MON_PROD_SALES

CUSTOMER	**MONTH**	**PROD**	**QUANTITY**
Juan	1/98	25	5
Juan	1/98	7	7
Juan	2/98	4	6
Luis	1/98	100	8
Luis	4/98	100	3

Primitive P12.1 – De-normalized Hierarchy Generation

Description:

This primitive generates a hierarchy relation, having as source relations that include a hierarchy or a part of one. It also transforms the original relations, so that they do not include the hierarchy any more. Instead of this, they reference the new hierarchy relation through a foreign key.

Input:

- source schema: $R_1, ..., R_n$ / $\exists$ A / A $\in$ $Att_D(R_i)$, i= 1...n $\wedge$ A represents the lowest level in the hierarchy
- { $J_1, ..., J_m$ }, set of attributes that constitute a hierarchy / A $\in$ { $J_1, ..., J_m$ } $\wedge$ A is the lowest level
- K / K $\in$ { $J_1, ..., J_m$ } key for the hierarchy
- source instance : $r_1, ..., r_n$

Resulting schema:

- R' ($J_1, ..., J_m$) $\in$ Rel_J / { K } $\in$ $Att_K(R')$
- R'_i / $Att(R'_i)$ = { K } $\cup$ ($Att(R_i)$ - { $J_1, ..., J_m$ })

Generated instance:

r' :

for each i:1..n do
    s_i = select $Att(Ri) \cap \{J_1, ..., J_m\}$
        from R_i
s = Integrate($s_1, ..., s_n$)
Insert s in R'
For each i:1..m / $\forall$ j:1..n, $J_i \notin Att(R_j)$
    Fill values for J_i in R'

r'_i:

for each tuple t of r_i
    if K = A then
        t'.$Att(R'_i)$ = t.$Att(R'_i)$
    else
        t'.{$Att(R'_i)$ – K} = t.{$Att(R'_i)$ – K}
        t'.K = select K
            from R'
            where R'.A = t.A
    insert t' into r'_i

Example:

EMPLOYEES

SSN	NAME	POSITION	ADDRESS	REGION	CITY
2190882	R. Mendez	C1	Bvar. Artiga	P. Rodo	Montevideo
2233553	S. Nunez	C1	J. Herrera y	Centro	Montevideo
7657657	L. Lopez	C1	18 de Julio	Centro	Montevideo
3476434	M. Kiuyd	C2	21 de Setie	Pocitos	Montevideo
4567326	S. Sanchez	C2	Gral. Flores	Centro	Montevideo
4678893	W. Yan	C3	Gonzalo Ra	P. Rodo	Montevideo
4888640	B. Pitt	C3	Bvar. Españ	Pocitos	Montevideo

BRANCHES

CODE	NAME	ADDRESS	REGION	CITY	COUNTRY
C1	A	Bvar. Artiga	P. Rodo	Montevideo	Uruguay
C2	B	J. Herrera y	Centro	Montevideo	Uruguay
C3	C	19 de Julio 122	Centro	Bs. As.	Argentina
C4	D	Florida 3998	Palermo	Bs. As.	Argentina

We want to have the geographic hierarchy in only one table, which can be referenced from dimensions. This hierarchy will be extracted from the relations EMPLOYEES and BRANCHES.

We apply primitive **P12.1**, where the input is:

- R_1 = EMPLOYEES, R_n = BRANCHES, A = REGION
- { J_1, ..., J_m } = { GEO_COD, REGION, CITY, COUNTRY }
- K = GEO_COD
- r_1 = tuples of EMPLOYEES, r_2 = tuples of BRANCHES

Result:

GEOGRAPHICS

GEO_COD	REGION	CITY	COUNTRY
G01	P. Rodo	Montevideo	Uruguay
G02	Centro	Montevideo	Uruguay
G03	Pocitos	Montevideo	Uruguay
G04	Centro	Bs. As.	Argentina
G05	Palermo	Bs. As.	Argentina

EMPLOYEES

NSS	NAME	POSITION	ADDRESS	GEO_COD
2190882	R. Mendez	C1	Bvar. Artiga	G01
2233553	S. Nunez	C1	J. Herrera y	G02
7657657	L. Lopez	C1	18 de Julio	G02
3476434	M. Kiuyd	C2	21 de Setie	G03
4567326	S. Sanchez	C2	Gral. Flores	G02
4678893	W. Yan	C3	Gonzalo Ra	G01
4888640	B. Pitt	C3	Bvar. Españ	G03

BRANCHES

CODE	NAME	ADDRESS	GEO_COD
C1	A	Bvar. Artiga	G01
C2	B	J. Herrera y	G02
C3	C	19 de Julio 122	G04
C4	D	Florida 3998	G05

5. Designing DW through primitives: strategies and rules to apply them

When designing a DW schema through primitives, two aspects (related to the obtained schema) should be considered: (i) schema consistency and (ii) schema suitability for the DW requirements. We consider a schema consistent if it satisfies the invariants defined in section 3. The DW requirements, which usually consist on complex queries that imply large volumes of data, are the ones that determine the data structures that are the most convenient for the DW schema. A good design should structure data so that queries can be satisfied as efficiently as possible.

In order to help the designer in making a design that is suitable for the requirements of his DW, we provide a set of strategies, some of which are presented in the following section. Afterwards, we show some rules whose goal is to assure consistency of the generated schema.

5.1. DW Design Strategies

Strategies for application of primitives were designed having into account some typical problems of Data Warehousing and should be useful to solve them.

The strategies proposed address design problems relative to: dimension versioning, versioning of N:1 dimension relationships, data summarization and data crossing, hierarchies' management, and derived data. Due to space limitations, we present only one group of strategies. (For all the strategies provided refer to [9]).

S1 - Dimension versioning

Real-world subjects represented in dimensions, usually evolve through time. For example, a customer may change his address, a product may change its description or package_size. Sometimes it is required to maintain the history of these changes in the DW. In some of these cases it is necessary to store all versions of the element so that the whole history is maintained. In other ones, only a fixed number of values of certain attributes should be stored. For example, it could be useful to maintain the current value of an attribute and the last one before it, or the current value and the original one.

A usual problem DW designers have to face is how to manage dimension versioning. This refers to how dimension information must be structured when its history needs to be maintained. The idea is to maintain versions of each real-world subject information.

Several alternatives are provided. In all of them, a new dimension relation is generated, where historical data about the subjects can be maintained.

The following are the possible strategies to apply:

1) Apply **Temporalization** primitive (P3), such that the time attribute belongs to the key of the relation.

2) Generalize the key of the dimension relation through one of the primitives of **Key Generalization** family (P4). The two options are:

2.1) Apply **Version Digits** primitive (P4.1), so that version digits are added to the key.

2.2) Apply **Key Extension** primitive (P4.2). In this case new attributes of the relation are included in the key.

3) Add new attributes, so that a small number of versions of certain data can be maintained. Do this, applying the primitive **Attribute Adding** (P7).

4) Generalize the key of the relation following alternatives **2.1** or **2.2**, and add an attribute of time that does not belong to the key (P4.1, P3 or P4.2, P3).

5) Partition the relation according to its stability through one of the primitives of **Partition by Stability** (P11). Here the alternatives are:

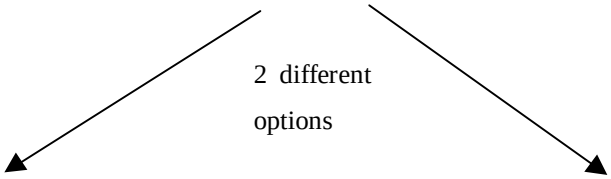
5.1) Vertically partition the relation, according to attribute values stability, through **Vertical Partition** primitive (P11.1).

5.2) Horizontally partition the relation, generating a relation for current data and another one for historical data, through **Horizontal Partition** primitive (P11.2). Immediately apply alternatives **1**, **2** or **4** to the history relation generated.

The following is an example showing application of some of these alternatives:

CUSTOMERS

SSN	NAME	AGE	INCOME	ADDRESS	SEX	CITY	MS
276052	R. Mendez	20	10000	Bvar. Artigas 3	F	Montevideo	S
342587	S. Nunez	30	15000	J. Herrera y Ob	M	Montevideo	C
431222	M. Garcia	20	10000	Garzon 2125	F	Salto	S
213438	L. Lopez	50	5000	18 de Julio 643	M	Colonia	C



CUSTOMERS_1

GRAL_SSN	NAME	.....
01276052	R. Mendez	.....
01342587	S. Nunez	.....
01431222	M. Garcia	.....
01213438	L. Lopez	.....

CUSTOMERS_2

SSN	DATE	NAME	.....
276052	1/1/93	R. Mendez	.....
342587	23/4/97	S. Nunez	.....
431222	5/2/98	M. Garcia	.....
213438	3/3/99	L. Lopez	.....

5.2. Consistency Rules

These are some rules that should be applied always, when a DW schema is being constructed through application of the primitives.

The rules consider the different cases of inconsistencies that can be generated by application of primitives and state the actions that must be performed to correct them.

R1 – Foreign key updates

R1.1 –

ON APPLICATION OF: *Temporalization* (adding the time attribute to the key) or *Key Generalization* to R, where X = old key and Y = new key

APPLY: *Foreign Key Update* to all R_i / $Att_{FK}(R_i,R)$ = X, obtaining $Att_{FK}(R_i,R)$ = Y

R1.2 –

ON APPLICATION OF: *Vertical Partition* to R with key X, obtaining R_1, R_2, R_3, with key X for each case

APPLY: *Foreign Key Update* to all R_i / $Att_{FK}(R_i,R)$ = X, obtaining $Att_{FK}(R_i,R_1)$ = X, $Att_{FK}(R_i,R_2)$ = X, $Att_{FK}(R_i,R_3)$ = X

R2 – Measure relations correction

ON APPLICATION OF: *Data Filter or Aggregate Generation* to R ∈ Rel_M removing A ∈ $Att_D(R)$, obtaining relation R'

WHEN: ∃ S ∈ Rel_D / $Att_{FK}(R', S) = \emptyset \wedge \exists$ B / B ∈ $Att(R') \wedge$ B ∈ $Att(S)$

APPLY: *Data Filter* to R' removing attribute B

R3 – History relations update[3]

ON APPLICATION OF: *DD-Adding*, *Attribute Adding*, *Hierarchy Generation*, *Aggregate Generation* or *Data Array Creation* to R, adding A / A ∈ $Att(R)$

WHEN: ∃ R' / R' ∈ $Rel_H(R)$

APPLY: *Attribute Adding* to R', obtaining A ∈ $Att(R')$

6. Conclusion

This paper presents a set of schema transformation primitives as well as some strategies and rules for their practical application. These schema primitives enable to design a DW from a source schema acting as design building blocks that have DW design knowledge embedded in their semantics.

In addition, the primitives provide a trace of the design, which is of great importance for documentation and design process management. Furthermore, the primitives enable to cope with source schema evolution in a DW context. Specifically, in DWs with highly evolutive source databases (e.g., extracting data from the Web [10],[11]) they enable to perform the repercussion of the source schema changes to the DW.

In our proposition schema consistency is managed through DW schema *invariants* and *rules*. While *invariants* specify the consistency conditions the DW schemas must satisfy, the *rules* state additional schema transformations to maintain the DW schema in a consistent state.

The main contribution of this paper is the proposition of a set of DW design primitives complemented with strategies and rules for their application. We believe that is a step forward the definition of well structured methodologies for DW design and their implementation in CASE tools.

Concerning the scope of the proposed primitives, the presented design strategies show how a wide spectrum of DW design problems can be solved through application of primitives.

---

[3] This rule is optional

On going work covers different topics: experimentation with the primitives by applying them in real DW developments [12], focus in the use of the primitives to trace the design and cope with the repercussion of source schema evolution, and specification of the instance transformations.

Besides, we are currently implementing the primitives in a DW design tool that enables the designer to apply the primitives through a graphical interface [13]. In addition this tool would include design guidelines based on the defined strategies and would implement the correction rules.

The current version of primitives does not include schema integration facilities. We consider that this is a problem itself, which involves specific aspects like concept correspondence specification, conflict resolution, schema merging, etc. Nevertheless we believe that the primitives should enable to perform schema integration in some way.

References

[1] Batini, Ceri, Navathe. *Conceptual Database Design. An Entity-Relationship Approach.* The Benjamin/Cummings Publishing Company, Inc. 1992

[2] J. L. Hainaut. *Entity-Generating schema transformations for Entity-Relationship models.* ER 1991: 643 – 670.

[3] B. Staudt Lerner, A. Nico Habermann. *Beyond Schema Evolution to Database Reorganization.* ECOOP/OOPSLA 1990 Proceedings.

[4] R. Kimball. *The Data Warehouse Toolkit.* J. Wiley & Sons, Inc. 1996

[5] R. Kimball. *The Data Warehouse Lifecycle Toolkit.* J. Wiley & Sons, Inc. 1998

[6] R. Kimball. *Data Warehouse Architect.* Column in DBMS online magazine. 1997

[7] C. Adamson, M. Venerable. *Data Warehouse Design Solutions.* J. Wiley & Sons, Inc. 1998

[8] L. Silverston, W. H. Inmon, K. Graziano. *The Data Model Resource Book.* J. Wiley & Sons, Inc. 1997

[9] A. Marotta. *A transformations based approach for designing the Data Warehouse.* Technical Report. Department of Computer Science of Engineering Faculty. Montevideo – Uruguay.

[10] R. Hackathorn. *Reaping the Web for your Data Warehousing.* DBMS, August 1998.

[11] R. D. Hackathorn. *Web Farming for the Data Warehouse.* Morgan Kaufmann Publishers, Inc. San Francisco, California. 1999

[12] R. Abella, L. Coppola, D. Olave,. *A Datawarehouse for the Engineering Faculty.* Graduate Project of the Engineering Faculty – Montevideo – Uruguay. 1998.

[13] P. Garbusi, F. Piedrabuena, G. Vazquez. *Design and Implementation of a schema transformation based DW design tool.* Graduate Project of the Engineering Faculty – Montevideo – Uruguay. On going work 1999.