Designing relational data warehouses through schema-transformation primitives[‡]

Adriana Marotta

Instituto de Computación, Facultad de Ingenieria, Universidad de la República, Montevideo, Uruguay

December 2000

Abstract

A Data Warehouse (DW) is a database that stores information oriented to satisfy decision-making requests. It is a database with some particular features concerning the data it contains and its utilisation .The features of DWs cause the DW design process and strategies to be different from the ones for OLTP Systems. We address the DW Design problem through a schema transformation approach. We propose a set of schema transformation primitives, which are high-level operations that transform relational sub-schemas into other relational sub-schemas. We also provide some tools that can help in DW design process: (a) the design trace, (b) a set of DW schema invariants, (c) a set of rules that specify how to correct schema-inconsistency situations that were generated by applications of primitives, and (d) some strategies for designing the DW through application of primitives.

Keywords

Data Warehouse (DW), DW design, schema transformation, Relational DW, DW design trace

[‡] This work was supported by Comisión Sectorial de Investigación Científica from Universidad de la República, Montevideo, Uruguay

1. Introduction

A Data Warehouse (DW) is a Database that stores information oriented to satisfy decision-making requests. A very frequent problem in enterprises is the impossibility for accessing to corporate, complete and integrated information of the enterprise that can satisfy decision-making requests. A paradox occurs: data exists but information cannot be obtained. In general, a DW is constructed with the goal of storing and providing all the relevant information that is generated along the different databases of an enterprise.

A DW is a database with some particular features. Concerning the data it contains, it is the result of transformations, quality improvement and integration of data that comes from operational bases. Besides, it includes indicators that are derived from operational data and give it additional value. Concerning its utilisation, it is supposed to support complex queries (summarisation, aggregates, crossing of data), while its maintenance does not suppose transactional load. In addition, in a DW environment end users make queries directly against the DW through user-friendly query tools, instead of accessing information through reports generated by specialists.

In this work we concentrate in DW design. The data model considered in this work is the Relational Model, for both the DW and the source databases.

The features of DWs cause the DW design process and strategies to be different from the ones for OLTP[1] Systems [Kim96-1]. For example, in DW design, the existence of redundancy in data is admitted for improving performance of complex queries and it does not imply problems like data update anomalies, since data is not updated on-line (DWs' maintenance is performed by means of controlled batch loads). Another issue to be considered is that a DW design must take into account not only the DW requirements, but also the features and existing instances of the source databases.

Existing knowledge related to this area is addressed in [Gut00].

The goal of this work is to provide a help tool that allows designing a DW starting from the source database and propagating source schema evolution to the DW.

We address the DW Design problem through a schema transformation approach. We propose a set of schema transformation primitives, which are high-level operations that transform relational sub-schemas into other relational sub-schemas. The idea for the design process is that the designer, taking into account the DW requirements and his own design criteria, applies primitives to construct a DW schema from a source schema.

We design the primitives considering the set of schema structures that are the most used in relational DWs and the possible existing source structures, so that there is one primitive for each one of these target and source structures.

[1] OLTP: On Line Transaction Processing

Having the primitives as the core of the proposal for DW design, we also provide some tools that help in DW design process. The first is the design trace, which is generated when a DW schema is constructed through application of primitives. The second is a set of schema invariants. Schema invariants are properties useful to check DW schema consistency. Having these invariants, we provide a set of rules that specify how to correct schema-inconsistency situations that were generated by applications of primitives. Finally, we provide some strategies for designing the DW through application of primitives. These strategies serve as guidelines for solving some common DW design problems.

The main contribution of this work is the proposal of a set of DW schema design primitives. These primitives must be applied to the source schema. 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.

The main interest for the definition of design primitives is twofold. First, primitives materialise 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 paper consists of 8 sections. Section 2 presents the framework we propose for DW logical design, Section 3 presents a set of DW-schema Invariants, Section 4 presents the Schema Transformation Primitives. In Section 5 and 6 Consistency Rules and Design Strategies for application of the primitives are proposed, and in Section 7 the Design Trace is presented. Finally, Section 8 presents the conclusions and future work.

2. Data Warehouse logical design framework

One of the most important tasks in the construction of a DW is the logical design of its schema. This logical design has to be done considering the particularities a DW has with respect to the information it stores and the requirements it has to support (described in Chapter 1). The techniques that are used for designing a database of an OLTP system are not applicable for designing a DW [Kim96-1], due to the existing differences between these two kind of databases.

We propose a tool that is intended to be of help at the time of designing a DW. Together with this tool we provide some guidelines for its utilisation. The tool is a set of **schema transformation primitives** that must be applied to a source schema in order to obtain a corresponding DW schema. The designer has to use his own design criteria to apply the primitives, although we give him some help through a set of rules and strategies he can use.

The primitives work with one source schema; they are not useful for performing integration of several source schemas. In this work we assume that the design process starts from an integrated schema.

Figure 3.1 shows the basic architecture of the transformation of a source schema into a DW schema, through the application of primitives.

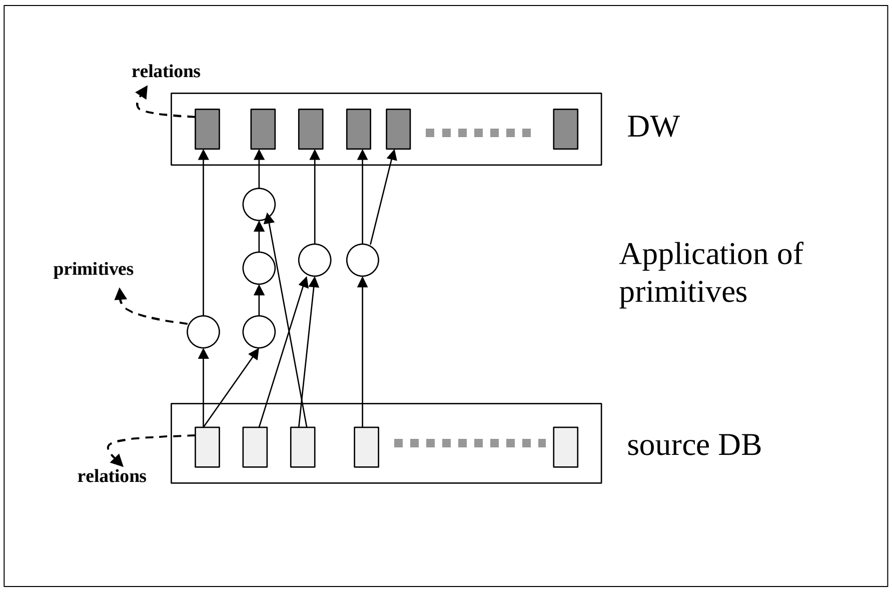


Figure 3.1. Application of primitives

In our approach, DW design is a process that starts with a source database schema, applies transformations to it, and ends with a resulting DW schema. The transformations are applied through the primitives to the source schema and to the intermediate sub-schemas[2] that are generated during the process, i.e. primitives are composed to obtain the final schema. Therefore, all the elements that constitute the final schema, are results of primitives application to the source schema.

The primitives are high-level transformations of Relational schemas. Roughly speaking, they take as input a sub-schema and their output is another sub-schema. Besides, they include an outline of the transformations that have to be applied to the source instance. 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.

For ensuring schema consistency we provide: (i) a set of DW schema invariants, and (ii) a set of consistency rules for application of primitives. We consider a schema consistent if it satisfies the DW schema invariants we define. With (i) we can check the consistency of the DW schema. (ii) states the actions that must be done on application of certain primitives in certain situations, with the form of ECA (Event Condition Action) rules, in order to preserve schema consistency.

[2] We consider a sub-schema as a set of relations that are part of a schema.

In addition, for assisting the designer during the design process, we provide strategies for solving typical problems that appear in DW design. These strategies should act as guidelines for the application of the primitives, covering many possible design alternatives for each considered design problem. Note that the primitives themselves do not lead to some specific strategy or methodology. Moreover, their application without well-defined design criteria, could lead to undesired results. For us, a good design should structure data so that the DW requirements can be satisfied efficiently. 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.

In the following sub-section we present some basic definitions about the model we use for the specification of the primitives.

In the Appendix we present an example of a complete design process through primitive applications.

2.1. 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 following are the sets defined over the Relational Model:

Relation[3] 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_C – Set of "crossing" 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_M – Set of "measure" relations. These are the crossing relations that have at least one 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 there exists a hierarchy among a set of attributes, can only be determined having into account the semantics of them.

Rel_H – Set of "historical" relations. These are the relations that have historical information that corresponds to information in other relation.

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

We define a function $f_H : Rel_H \rightarrow Rel$, which, given a historical relation, returns the corresponding current relation.

These sets verify the following properties:

- $Rel_M \subset Rel_C$
- $Rel_J \subset Rel_D$
- $Rel_H \subset (Rel_D \cup Rel_C)$

<u>Attribute sets:</u>

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 is a 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: $X \cap Y \neq \emptyset$

The following are some definitions that are necessary for the specifications we present in the rest of the document.

Rel_Name – Set of relation names.
Att_Name – Set of attribute names.
Primitive_Name – Set of primitive names.
Fun_Name – Set of function names.

subst (A, B, X) – function that substitutes attribute A by attribute B in the set of attributes X.
conc (s1, s2) – function that concatenates two strings.
name (A) – function that returns the name of an attribute.

3. DW schema Invariants

Considering the classification we have defined for the elements of a DW schema, we can find some conditions that must be satisfied by the different types of elements, in order to maintain the consistency in the DW schema.

In this section we propose a set of DW schema invariants. They are a set of properties that must be satisfied by a relational DW schema in order to be consistent.

Invariants:

1. **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 = $Att_{FK}(R_1, R_2)$, it holds: X $\in$ $Att_K(R_2)$ $\wedge$ $\forall$ R / X $\in$ $Att_K(R)$, X $\in$ $Att_{FK}(R_1, R)$

2. **Hierarchies :**

   Given a set of attributes X representing a hierarchy, a functional dependency must hold between each attribute of X and all attributes of X that identify 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 identifies a higher level in the hierarchy than "a"

   it holds $A_1 \rightarrow A_2$

   $A_2 \rightarrow A_3$

   ....

   $A_{n-1} \rightarrow A_n$

3. **History relations :**

   - A history relation that corresponds to a current data relation, 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 = $Att_{FK}(R_H,R)$

4. **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 Schema Transformation Primitives

In this section we propose a set of schema transformation primitives. Our goal is to provide a set of high-level transformations that can be combined to cover a wide spectrum of DW schema designs. The idea is that these transformations are applied to a schema in order to make it more suitable for the kind of queries that will be submitted to it.

The kind of transformations involved by the primitives are: table partitions, table merges, attribute addings, attribute removes, and keys and foreign keys changes.

Figure 3.2 shows a table containing the whole set of primitives proposed. In this table, the primitive names marked with a "*" symbol correspond to groups of primitives.

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 crossing in a new relation.

Figure 3.2. The set of primitives

As seen, the proposed schema transformation primitives do not intend to be "complete" in the sense of enable the design of any Relational schema, but they are intended to enable the design of DW. We find there is a trade-off between the level of expressiveness and the compactness of the set of primitives.

The following sub-sections present: first the description of all primitives and second the specifications of them.

4.1. Descriptions of primitives

This section presents a description of each primitive. These descriptions are intended to show the usefulness of the primitives as well as their behaviour.

Primitive 1. IDENTITY

This primitive is useful when we want to generate in the DW, a relation that is exactly the same as another one. The original relation may be one existing in the source database or one that is an intermediate result (the result of the application of a primitive).

It gives as result, a copy of the relation given as input.

Primitive 2. DATA FILTER

In operational databases, there are some attributes that are of interest for the DW system, but there are some others that correspond to data that is purely operational and that is not useful for the kind of analysis that is made with the DW.

The goal of this primitive is to preserve only the useful attributes, removing the other ones.

Primitive 3. TEMPORALIZATION

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.

In DWs, many relations need to include a temporal element, so that they can maintain historical information.

This primitive adds an element of time to the set of attributes of a relation.

Primitive 4. KEY GENERALIZATION

The real world subjects represented in dimensions, usually evolve through time. For example, a client may change his address, a product may change its description or package_size.

In some cases it is enough to maintain only the last value, but in other ones it is necessary to store all versions of the element, so that history is maintained.

The goal of this group of primitives is to generalize the primary key of a dimension relation, so that more than one tuple of each subject represented in the relation, can be stored.

Two alternatives are provided to do this generalization, through the primitives: Version Digits and Key Extension.

<u>Primitive 4.1.</u> VERSION DIGITS

To generalize the key, version digits are added to each value of the attribute.

<u>Primitive 4.2.</u> KEY EXTENSION

The key is extended; new attributes of the relation are included in it.

<u>Primitive 5.</u> FOREIGN KEY UPDATE

When the key of a relation is changed, it is necessary to make the same changes in all the foreign keys that reference to it from other relations. For example, if an attribute is added to a key, it must be added also to the foreign keys of the referencing relations.

This primitive is useful for updating a foreign key in a relation when its corresponding primary key is modified.

<u>Primitive 6.</u> DD-ADDING

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 these kind of data calculated, for performance reasons.

The primitives of this group add an attribute that is derived from others to a relation. They never cause changes to the grain of the relation.

<u>Primitive 6.1.</u> DD-ADDING 1-1

In this case, the calculations are made over only one relation and one tuple. For example, the total import of a sale is calculated from the quantity sold and the unit-price, which are all in the same relation.

Primitive 6.2. DD-ADDING N-1

In this case two relations are used. A calculated attribute is added to one of the relations. This attribute is derived from some attributes from the same relation and others from the other relation. This is the case of the example mentioned for the group of primitives (product prices).

The calculation function works over only one tuple of the relations. This tuple must be obtained uniquely through a join of the two relations.

Primitive 6.3. DD-ADDING N-N

This is the more complex case. Two relations and n tuples are used for the attribute calculation. Consider the following example in a bank. There exists a relation with client data and another relation with account data. If we want to add to the former the total amount of all the accounts for each client, the amounts contained in the second relation must be summed for each client.

The calculation function works over a set of tuples of one of the relations. These tuples must be obtained through a join of the two relations.

Primitive 7. ATTRIBUTE ADDING

The real world subjects represented in dimensions usually evolve through time. For example, a client 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 cases, only a fixed number of values of certain attribute 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 of an attribute and the original one.

In these cases, empty attributes are reserved in a dimension relation, for future changes. Suppose, for instance, that when a client changes his address we want to store the new and the old addresses. With this primitive an attribute is added to the relation, initially with a null value, to be filled in case the client moves out.

Primitive 8. HIERARCHY ROLL UP

In operational databases the information in the relations are stored at the highest level of detail that is possible. For example, the measure relations use to have all the movements. Usually, in these relations there is an attribute that has a hierarchy associated.

Often, when these relations are used in a DW, they are summarised by an attribute following some hierarchy (doing a "roll-up"), for example, if data is in a daily level and monthly totals are required. In this case we are doing a roll-up in a hierarchy of time.

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.

Primitive 9. AGGREGATE GENERATION

In operational systems data is managed as crossings of many dimensions. In general, many DW relations are constructed from these crossings, and data is grouped by some of the dimensions. Other dimensions are removed as a consequence of this information grouping.

For example, for a salary system, may be of great importance which employee has made certain sale. However, for analysing the sales at a global level in the DW, it is required resumed data and not that information in particular.

This primitive removes a set of attributes from a measure relation, summarising the measures. This operation has the effect of decreasing the number of tuples of the relation.

Primitive 10. DATA ARRAY CREATION

In a relation where measures are maintained on a month-by-month basis, it can be useful, instead of having an attribute for the month and another one for the measure, to have 12 attributes for the measures of the 12 months respectively. With this structure comparative reports can be done more easily and with better performance, since annual totals are calculated at a tuple level. Besides, the number of tuples decreases.

This multiple attributes schema (data array) is useful not only for months, in fact it can be used for any attribute whose associated set of values is finite and known (so that an attribute can be assigned to each value).

Given a relation that contains an attribute that represents a pre-determined set of values, this primitive generates a relation with a data array structure.

Primitive 11. PARTITION BY STABILITY

In some cases it is recommended to partition a relation, distributing its data into different relations. This can be useful, for example, for maintaining the most recent data more accessible than the rest of the data. It also allows organising data according to its propensity for change.

These primitives partition a relation, in order to organise its data storage. The first (Vertical Partition) or the second primitive (Horizontal Partition) of this family, can be applied, depending on the design criterion used.

Primitive 11.1. VERTICAL PARTITION

This primitive applies a vertical partition to a dimension relation, giving several relations as result. It distributes the attributes, so that they are grouped according to their propensity for change.

Primitive 11.2. HORIZONTAL PARTITION

Two relations, one for more current data and the other for historical information, are generated from an original one. Each resulting relation contains the same attributes as the source one.

Primitive 12. 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.

In addition, they transform the original relations, so that they do not include the hierarchy any more. Instead of this, they reference the new hierarchy relation or relations, through a foreign key.

The three primitives that compose this family implement three different design alternatives for the generated hierarchy.

Primitive 12.1. DE-NORMALISED

This primitive generates only one relation for the hierarchy.

Primitive 12.2. SNOWFLAKE

This primitive generates several relations for the hierarchy, representing it in a normalised form.

Primitive 12.3. FREE DECOMPOSITION

This primitive generates several relations for the hierarchy. The form (distribution of attributes) of these relations is decided by the designer.

Primitive 13. MINIDIMENSION BREAK OFF

Often, in a dimension there is a set of attributes that have a limited number of possible values.

The idea is to code the various combinations of values of these attributes (only the combinations that really occur) and store them in a separate relation, so that they can be referenced from other relations. Storage space is saved using this structure.

This primitive generates two dimension relations. One is the result of eliminating a set of attributes from a dimension relation. The other is a relation that contains only this set of attributes. Besides, it defines a foreign key between the two relations.

<u>Primitive 14.</u> NEW DIMENSION CROSSING

In many cases, we need to materialise a dimension crossing in a new relation. This can be done through a join of some relations. For example, there is a measure relation where the product dimension is crossed with other dimensions, and another relation where supplier is determined by product. The supplier dimension can be added to the measure relation and the product can be removed, obtaining a crossing between supplier and the other dimensions existing in the measure relation.

4.2. Specifications of primitives

The following specifications present four sections. The **Description** specifies a natural language description about the primitive behaviour. The **Input** specifies the source schema and other arguments that are necessary for the application of the primitive. The **Resulting schema** is the specification of the schema that is generated by the primitive. The **Generated instance** is a sketch of the transformation that has to be applied to the instance of the source schema in order to populate the generated schema.

<u>Primitive 1.</u> IDENTITY

Description:

Given a relation, it generates another that is exactly the same as the source one.

Input:

- source schema : R $\in$ *Rel*
- source instance : r

Resulting schema:

- R’ $\in$ *Rel* / R’ = R

Generated instance:

```
▪ r’ =   select *
         from R
```

Primitive 2. **DATA FILTER**

Description:

Given a source relation, it generates another one where only some attributes are preserved. Its goal is to eliminate purely operational attributes.

Input:

- source schema : $R (A_1, ..., A_n) \in \mathit{Rel}$
- $X \subset \{ A_1, ..., A_n \} \wedge X \subset \mathit{Att}_D(R)$
- source instance : r

Resulting schema:

- $R' (A'_1, ..., A'_m) \in \mathit{Rel}$ / $\{ A'_1, ..., A'_m \} = \{ A_1, ..., A_n \} – X$

Generated instance:

- r' = select $A'_1, ..., A'_m$
  from R

Primitive 3. **TEMPORALIZATION**

Description:

It adds an element of time to the set of attributes of a relation.

Input:

- source schema : $R (A_1, ..., A_n)$ / $\exists X \subset \{ A_1, ..., A_n \} \wedge X \in \mathit{Att}_K(R)$
- T time attribute / $DOM(T) = \{ t_0, ..., t_k \}$ set of time measures $\vee$
  $DOM(T) = \{ c / c \subseteq \{ t_0, ..., t_k \}$ set of time measures $\}$.
- Key, Boolean argument. It tells if T will be part of R's key or not.
- source instance : r

Resulting schema:

- $R' (A_1, ..., A_n, T)$ / $T \in \mathit{Att}_D \wedge$ if key then $XT \in \mathit{Att}_K(R)$

Generated instance:

- r' = select $A_1, ..., A_n$, V(t)
  from R

where V(t) is a user-function. It gives, for example, the snapshot time or snapshot date.

Primitive 4. **GROUP: KEY GENERALIZATION**

Description:

These primitives generalise the primary key of a dimension relation, so that more than one tuple of each subject represented in the relation can be stored.

Primitive 4.1. **VERSION DIGITS**

Description:

To generalise the key, version digits are added to each value of the attribute.

Input:

- source schema : R (A_1, ..., A_n) ∈ Rel_D / A_1 ∈ $Att_K(R)$
- source instance : r

Resulting schema:

- R' ∈ Rel_D / *Att(R') = subst* (A_1, B, *Att(R)*) , where *name*(B) = *conc* ('GR', *name*(A_1))

Generated instance:

- r' = select concat(num_gen, A_1), ..., A_n
  from R

where num_gen is a user-function that must generate series of numbers.

Primitive 4.2. **KEY EXTENSION**

Description:

The key is extended; new attributes of the relation are included in it.

Input:

- source schema : R (A_1, ..., A_n) ∈ Rel_D / ∃ X ⊂ { A_1, ..., A_n } ∧ X ∈ $Att_K(R)$
- Y ⊂ ({ A_1, ..., A_n } – X) , attributes to be added to the key
- source instance : r

Resulting schema:

- R' (A_1, ..., A_n) ∈ Rel_D / XY ∈ $Att_K(R')$

Generated instance:

- r' = r

Primitive 5. **FOREIGN KEY UPDATE**

Description:

Through this primitive, a foreign key and its reference can be changed in a relation.

Input:

- source schema : R (A_1, ..., A_n) ∈ *Rel* / X ∈ $Att_{FK}(R)$
- X, set of attributes to be eliminated
- Y, set of attributes which will substitute X
- { R_1, ..., R_m } set of relations with respect to which Y will be a foreign key
- S ∈ *Rel* / *Att(S)* = X ∪ Y , auxiliary relation that contains the correspondence between the old key and the new key
- Source instance : r, s

Resulting schema:

- R' ∈ *Rel* / *Att(R')* = Y U ({ A_1, ..., A_n } – X) ∧ Y = $Att_{FK}(R',R_1)$ ∧ ....
  ∧ Y = $Att_{FK}(R',R_m)$

Generated instance:

- r' = select Y ∪ ({A_1, ..., A_n} – X)
  from R S
  where R.X = S.X

Primitive 6. **GROUP: DD-ADDING**

Description:

The primitives of this group add to a relation an attribute that is derived from others.

In this kind of problem, four different cases can be distinguished taking into account the number of relations and the number of tuples that participate in the calculation.

t u p l e s

rels		1	n
	1	P 4.1	P 8
	n	P 4.2	P 4.3

In this group of primitives three primitives are proposed, which solve the cases of:

1 relation, 1 tuple

n relations, 1 tuple

n relations, n tuples

In these cases the derived attribute has the same grain as the other attributes of the relation.

The case of: *1 relation, n tuples*, is in essence different from the other ones because in the resulting relation the original grain is changed, eliminating some attributes and adding others. The goal in this case is to group information by certain attributes, which is different form the goal in the other cases. There are two separate primitives that treat this case: **Primitive 8 – Hierarchy Roll Up** and **Primitive 9 – Aggregate Generation**.

<u>Primitive 6.1.</u> **DD-ADDING 1-1**
Description: Given a relation, this primitive adds an attribute that is derived from others of the same relation.
Input: ▪ source schema : R (A_1, ..., A_n) ∈ *Rel* ▪ f (A_{i1}, ..., A_{im}) / { A_{i1}, ..., A_{im} } ⊆ { A_1, ..., A_n } , where f is a user-defined function ▪ source instance : r
Resulting schema: ▪ R’ (A_1, ..., A_n, A_{n+1}) ∈ *Rel* / A_{n+1} represents f (A_{i1}, ..., A_{im})
Generated instance: ▪ r’ = select A_1, ..., A_n, f (A_{i1}, ..., A_{im}) from R

Example:

DETAIL

ART. NUM.	QUANTITY	UNIT_PRICE
100	20	200
105	7	115
108	32	40

We want to have the total price calculated and materialised in the relation.
Primitive 6.1 is applied, where the input is:

▪ R = DETAIL
▪ f = QUANTITY x UNIT_PRICE
▪ r = tuples of DETAIL

<u>Result:</u>

DETAIL

ART. NUM	QUANTITY	UNIT_PRICE	TOTAL_PRICE
100	20	200	4000
105	7	115	805
108	32	40	1280

♦

Primitive 6.2. **DD-ADDING N-1**

Description:

This primitive adds to a relation an attribute that is derived from some attributes from the same relation and others from the other relation. In this case the calculation function works over only one tuple of the relations. This tuple must be uniquely obtained through a join operation. Besides, the derived attribute can be defined as a foreign key to another relation.

Note: This primitive works with only two relations. If participation of more than two relations is required, additional steps must be applied.

Input:

- source schema : R_1 (A_1, ..., A_n), R_2 (B_1, ..., B_m) $\in$ *Rel*
- f (C_1, ..., C_k) / { C_1, ..., C_k } $\subseteq$ { A_1, ..., A_n }$\cup${ B_1, ..., B_m }, where f is a user-defined function
- A / A $\in$ { A_1, ..., A_n } $\wedge$ A $\in$ { B_1, ..., B_m } , join attribute
- is_fk , Boolean argument (declare A_{n+1} as a foreign key or not)
- $R_3 \in$ *Rel* , relation to which A_{n+1} is a foreign key (optional)
- source instance : r_1, r_2

Resulting schema:

- R'_1 (A_1, ..., A_n, A_{n+1}) $\in$ *Rel* / A_{n+1} represents f (C_1, ..., C_k) $\wedge$

  if is_fk then $A_{n+1} = Att_{FK}(R'_1, R_3)$

Generated instance:

- r'_1 = select A_1, ..., A_n, f (C_1, ..., C_k)
  from R_1 R_2
  where R_1.A = R_2.A

Example:

PRODUCTS

PROD_COD	PROD_NAM	PRICE	SUPP_COD
C1	Clavos	5	P1
C2	Tornillos	3	P1
C3	Sillas	200	P14

SUPPLIERS

SUPP_COD	SUPP_NAM	ADDRESS	PHONE
P1	T&F	B. Artigas 444	121212
P14	Muebles Garcia	G. Flores 2255	545454

We want to have the supplier name in the PRODUCTS relation.
Primitive 6.2 is applied, where the input is:

- R_1 = PRODUCTS, R_2 = SUPPLIERS

▪ f = SUPPLIERS.SUPP_NAM
▪ A = SUPP_COD
▪ is_fk = FALSE
▪ r_1 = tuples of PRODUCTS, r_2 = tuples of SUPPLIERS

Result:

PRODUCTS

PROD_COD	**PROD_NAM**	**PRICE**	**SUPP_COD**	**SUPP_NAM**
C1	Clavos	5	P1	T&F
C2	Tornillos	3	P1	T&F
C3	Sillas	200	P14	Muebles Garcia

Note: For totally de-normalising, apply successively this primitive in the same fashion, adding the rest of the attributes of the relation SUPPLIERS.

♦

Primitive 6.3. **DD-ADDING N-N**

Description:

This primitive adds to a relation an attribute that is derived from an attribute of another relation. In this case the calculation function works over a set of tuples of the other relation. This set is obtained through a join operation between the two relations.

Note: This primitive works with only two relations. If participation of more than two relations is required, additional steps must be applied.

Input:

▪ source schema : R_1 (A_1, ..., A_n), R_2 (B_1, ..., B_m) $\in$ *Rel*
▪ e(B) / B $\in$ { B_1, ..., B_m } , where e(B) is an aggregate expression over the attribute B
▪ X / X $\subset Att_D(R_2)$, attributes by which we want to group
▪ A / A $\in$ { A_1, ..., A_n } $\wedge$ A $\in$ { B_1, ..., B_m } , join attribute
▪ source instance : r_1, r_2

Resulting schema:

▪ R'_1 (A_1, ..., A_n, A_{n+1}) $\in$ *Rel* / A_{n+1} represents e(B) in R_2

Generated instance:

▪ r'_1 = select A_1, ..., A_n, e(B)
from R_1 R_2
where R_1.A = R_2.A
group by A_1, ..., A_n, X

Example:

CUSTOMERS

SSN	**NAME**	**ADDRESS**	**PHONE**	**CS**
2760527	Juan Perez	B. Artigas 444	121212	S
5321532	Maria Lopez	G. Flores 2255	545454	C

ACCOUNTS

SSN	**ACCOUNT_NUM**	**AMOUNT**
2760527	15382130	5000
2760527	30010011	200
2760527	10001000	30000
5321532	15482122	12000
5321532	10001001	700

We want to have the total amount of money that each customer has in the bank.
Primitive 6.3 is applied, where the input is:

- R_1 = CUSTOMERS, R_2 = ACCOUNTS
- e(B) = SUM(AMOUNT)
- X = { SSN }
- A = SSN
- r_1 = tuples of CUSTOMERS, r_2 = tuples of ACCOUNTS

Result:

CUSTOMERS

SSN	**NAME**	**ADDRESS**	**PHONE**	**CS**	**AMOUNT**
2760527	Juan Perez	B. Artigas 444	121212	S	35200
5321532	Maria Lopez	G. Flores 2255	545454	C	12700

♦

Primitive 7. **ATTRIBUTE ADDING**

Description:

Given a dimension relation, this primitive adds one or more attributes to it.

Input:

- source schema : R (A_1, ..., A_n) $\in$ Rel_D
- { B_1, ..., B_m }, attribute set
- source instance : r

Resulting schema:

- R’ (A_1, ..., A_n, B_1, ..., B_m) $\in$ Rel_D

Generated instance:

- r’ = select A_1, ..., A_n, ‘NULL’, ..., ‘NULL’
  from R

Primitive 8. **HIERARCHY ROLL UP**

Description:

Given a measure relation R_1 and a hierarchy relation R_2, this primitive does a roll up to R_1 by one of its attributes following the hierarchy in R_2 (by a foreign key that must exist from R_1 to R_2). Besides, it can generate another hierarchy relation with the corresponding grain.

Input:

- source schema :
  - $R_1 (A_1, ..., A_n) \in Rel_M \ / \ \exists A \in \{ A_1, ..., A_n \} \wedge \{A\} = Att_{FK}(R_1, R_2)$
  - $R_2 (B_1, ..., B_n) \in Rel_J \ / \ A \in \{ B_1, ..., B_n \} \wedge \{ A \} \in Att_K(R_2)$
- Z set of attributes / card(Z) = k (measures)
- $B \ / \ B \in \{ B_1, ..., B_n \} \wedge B \in Att_D(R_2)$ (chosen hierarchy level)
- $\{ e_1, ..., e_k \}$, aggregate expressions
- $X \ / \ X \subset \{ A_1, ..., A_n \} \wedge X \subset (Att_D(R_1) \cup Att_M(R_1))$ (they have a lower grain)
- $Y \ / \ Y \subset \{ B_1, ..., B_n \} \wedge Y \subset Att_D(R_2)$ (they have a lower grain)
- agg_h , Boolean argument (generate a new hierarchy or not)
- source instance : r_1, r_2

Resulting schema:

- $R'_1 (A'_1, ..., A'_m) \in Rel_M \ / \ \{ A'_1, ..., A'_m \} = \text{sust} [A, B, \{ A_1, ..., A_n \} - X]$ $\wedge \ Att_{FK}(R'_1) = Att_{FK}(R_1) - Att_{FK}(R_1, R_2)$
- If agg_h then
  $R'_2 (B'_1, ..., B'_m) \in Rel_J \ / \ \{ B'_1, ..., B'_m \} = \{ B_1, ..., B_n \} - Y \ \wedge$
  $\{ B \} \in Att_K(R'_2) \ \wedge$
  $Att_{FK}(R'_1, R'_2) = \{ B \}$
- Note: Note that the original hierarchy relation is not part of the resulting schema in any case of application of this primitive.

Generated instance:

- r'_1 = select ($\{ A'_1, ..., A'_m \}$- Z) $\cup$ $\{ e_1, …., e_k \}$
  from R_1 R_2
  where $R_1.A = R_2.A$
  group by $\{ A'_1, ..., A'_m \} - Z$
- r'_2 = select distinct $B'_1, ..., B'_m$
  from R_2

Example:

SALES

CUSTOMER	SALESMAN	DATE	PROD	CITY	QUANTITY
Juan	Pedro	1/1/98	25	Montevideo	2
Juan	Pedro	5/1/98	25	Montevideo	3
Juan	Pedro	8/1/98	7	Colonia	7
Juan	Maria	7/2/98	4	Montevideo	1
Juan	Laura	1/2/98	4	Maldonado	5
Luis	Pedro	3/1/98	100	Montevideo	2
Luis	Laura	5/1/98	100	Montevideo	6
Luis	Laura	8/4/98	100	Canelones	3

TIME

DATE	WEEK	MONTH	TRIMESTER	YEAR
1/1/98	1/98	1/98	1/98	1998
3/1/98	1/98	1/98	1/98	1998
5/1/98	2/98	1/98	1/98	1998
8/1/98	2/98	1/98	1/98	1998
1/2/98	6/98	2/98	1/98	1998
7/2/98	6/98	2/98	1/98	1998
8/4/98	14/98	4/98	2/98	1998

We want to have the sales' information grouped by month instead of by date. We scale two levels in the hierarchy of time.
Primitive 8 is applied, where the input is:

- R_1 = SALES,   A = DATE, foreign key
- R_2 = TIME,   A = DATE, relation key
- Z = { QUANTITY }, card(Z) = k = 1, measure attribute
- B = MONTH
- { e_1, ..., e_k } = { sum(QUANTITY) }
- X = ∅
- Y = { DATE, WEEK }
- agg_h = true
- r_1 = tuples of SALES, r_2 = tuples of TIME

Result:

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

TIME_MONTH

MONTH	TRIMESTER	YEAR
1/98	1/98	1998
2/98	1/98	1998
4/98	2/98	1998

♦

<u>Primitive 9.</u> AGGREGATE GENERATION

Description:

Given a measure relation, this primitive generates another measure relation, where data is 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'_1 = 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 10. **DATA ARRAY CREATION**

Description:

The source schema considered by this primitive is a relation that includes an attribute representing a set of predetermined values (e.g., month). The primitive generates a relation that includes an attribute for each predetermined value.

Input:

- source schema : $R(A_1, ..., A_n) \in \mathit{Rel}$ / $\exists B \in \{A_1, ..., A_n\} \wedge$ B represents a set of predefined values
- $A \in \mathit{Att(R)}$
- $\{V_1, ..., V_k\}$ set of attributes corresponding to each value of B
- source instance : r

Resulting schema:

- $R'(A'_1, ..., A'_m) \in \mathit{Rel}$ /

$$\{A'_1, ..., A'_m\} = \{A_1, ..., A_n\} - \{A, B\} \cup \{V_1, ..., V_k\}$$

Generated instance:

- r' =

```
host variables: X, A, B
X = Att(R) – {A, B}
next (R, cursor)
while not end(cursor) do
        quant_v = corresp_att (:B)
        if empty ( select *
                from R'
                where X = :X)  then
            insert into R' (X, quant_v) values (:X, :A)
        else
            update R' set quant_v = :A where X = :X
        next (R, cursor)
end.
```

where corresp_att is a user-defined function that given a value of attribute B, gives the name of the corresponding attribute in R'.

Example:

SALES

SALESMAN	**CITY**	**QUANTITY_ SOLD**	**YEAR**
Ana	Montevideo	20	1997
Ana	Canelones	3	1997
Ana	Rivera	7	1997
Pedro	Montevideo	44	1997
Pedro	Canelones	62	1997
Pedro	Rivera	9	1997
Pedro	Salto	40	1997
Ana	Montevideo	50	1998
Ana	Canelones	32	1998
Ana	Rivera	10	1998
Ana	Salto	15	1998
Pedro	Montevideo	112	1998
Pedro	Canelones	20	1998
Pedro	Rivera	9	1998
Pedro	Salto	20	1998

Primitive 10 is applied, where the input is:

- R = SALES,   A = QUANTITY_SOLD, B = CITY
- { V_1, ..., V_k } = { MON_QUAN, CAN_QUAN, RIV_QUAN, SAL_QUAN }
- r = tuples of SALES

Result:

SALES_BY_CITY

SALESMAN	**YEAR**	**MON_QUAN**	**CAN_QUAN**	**RIV_QUAN**	**SAL_QUAN**
Ana	1997	20	3	7	0
Ana	1998	50	32	10	15
Pedro	1997	44	62	9	40
Pedro	1998	112	20	9	20

♦

<u>Primitive 11.</u> **GROUP: PARTITION BY STABILITY**

Description:

These primitives partition a relation, in order to organise its history data storage. The first (Vertical Partition) or the second primitive (Horizontal Partition) of this family, can be applied, depending on the design criterion used.

Source schema:

- $R\ (A_1, ..., A_n) \in Rel_D\ /\ X \in Att_K(R)$

<u>Primitive 11.1.</u> **VERTICAL PARTITION**

Description:

This primitive applies a vertical partition to a dimension relation, giving several relations as result. It should distribute the attributes, so that they are grouped according to their propensity for change.

Input:

- source schema : the source schema defined for the group
- $Y \subseteq \{ A_1, ..., A_n \}$, attributes which values never change
- $Z \subseteq \{ A_1, ..., A_n \}$, attributes which values sometimes change
- $W \subseteq \{ A_1, ..., A_n \}$, attributes which values change very frequently
  $W \cap Y \cap Z = \varnothing$
- source instance : r

Resulting schema:

- if $Y \neq \varnothing$ then $R_1\ (XY) \in Rel_D\ /\ X \in Att_K(R_1)$
- if $Z \neq \varnothing$ then $R_2\ (XZ) \in Rel_D\ /\ X \in Att_K(R_2)$
- if $W \neq \varnothing$ then $R_3\ (XW) \in Rel_D\ /\ X \in Att_K(R_3)$

Generated instance:

- $r_1 = \Pi_{XY}\ r$
- $r_2 = \Pi_{XZ}\ r$
- $r_2 = \Pi_{XZ}\ r$

<u>Primitive 11.2.</u> **HORIZONTAL PARTITION**

Description:

Two relations, one for more current data, and the other for historical information, are generated from an original one. Each new relation contains the same attributes as the source one. One relation is defined as historical with respect to the other.

Input:

- source schema : the source schema defined for the group
- source instance : r

Resulting schema:

- $R_{Cur} = R \ / \ X \in \mathit{Att}_K(R_{Cur})$
- $R_{His} = R \ / \ X \in \mathit{Att}_K(R_{His}) \ \wedge \ R_{His} \in \mathit{Rel}_H(R_{Cur})$

<u>Note:</u> The primitive assigns to R_{His} the same key as to R_{Cur}. However, this should be changed when one of the primitives suitable for the problem of versioning (P3 or P4) are applied to R_{His}.

Generated instance:

- $r_{Cur} = r$
- $r_{His} = \varnothing$

<u>Primitive 12.</u> **GROUP: HIERARCHY GENERATION**

Description:

These primitives generate hierarchy relations, having as source relations that include a hierarchy or a part of one. In addition, they transform the original relations, so that they do not include the hierarchy any more. Instead of this, they reference the new hierarchy relation or relations, through a foreign key.

The three primitives that compose this family implement three different design alternatives for the generated hierarchy. The alternatives are: de-normalized, totally normalized (snowflake), or partitioned in a form that is given by the designer.

Source schema:

- $R_1, ..., R_n \ / \ \exists \ A \ / \ A \in \mathit{Att}_D(R_i)$, $i = 1...n \ \wedge$ A is the lowest level of a hierarchy

Primitive 12.1. DE-NORMALIZED HIERARCHY GENERATION

Description:

This primitive generates only one relation for the hierarchy.

Input:

- Source schema : the source schema defined for the group
- $\{ J_1, ..., J_m \}$, set of attributes that constitutes 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 / $Att(R'_i) = \{ K \} \cup (Att(R_i) - \{ J_1, ..., J_m \}) \wedge \{K\} = Att_{FK}(R'_i, R')$, i: 1..n

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 into R'
          For each i:1..m / $\forall$ j:1..n, $J_i \notin Att(R_j)$ do
               Fill values of $J_i$ in R'

▪ r'_i =  for each tuple t of $r_i$ do
              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
                              add t' to $r'_i$
```

Example:

EMPLOYEES

SSN	EMP_NAM	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

BRAN_CODE	**BRAN_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 Junio	Centro	Bs. As.	Argentina
C4	D	Calle A 334	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	**EMP_NAM**	**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

BRAN_CODE	**BRAN_NAME**	**ADDRESS**	**GEO_COD**
C1	A	Bvar. Artiga	G01
C2	B	J. Herrera y	G02
C3	C	19 de Junio	G04
C4	D	Calle A 334	G05

♦

Primitive 12.2. **SNOWFLAKE HIERARCHY GENERATION**

Description:

This primitive generates several relations for the hierarchy, representing it in a normalised form.

Input:

- source schema : the source schema defined for the group
- $J_1, ..., J_m$, sorted list of attributes that constitutes a hierarchy / $A \in \{ J_1, J_2 \} \wedge$ A is the lowest level
- K / $K = J_1$, key for the hierarchy
- source instance : $r_1, ..., r_n$

Resulting schema:

- $R_{Ji} (J_i, J_{i+1}) \in Rel_J \wedge J_i \in Att_K(R_{Ji}) \wedge J_{i+1} = Att_{FK}(R_{Ji}, R_{Ji+1})$, i: 1..m-1
- R'_i / $Att(R'_i) = \{ K \} \cup (Att(R_i) - \{ J_1, ..., J_m \}) \wedge \{K\} = Att_{FK}(R'_i, R_{J1})$, i: 1..n

Generated instance:

```
▪ r_J1, ...., r_Jm =
          for each i :1..n do
               s_i = select Att(Ri) ∩ {J_1, ...., J_m}
                    from R_i
          s = Integrate ( s_1, ...., s_n )
          Insert in snowflake mode, s into R_J1, R_J2, ...., R_Jm-1
          for each i:1..m / ∀ j:1..n, J_i ∉ Att(R_j) do
               Fill values of J_i in R_J1, R_J2, ...., R_Jm-1

▪ r'_i =  for each tuple t of r_i do
               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_J1
                              where R_J1.A = t.A
               add t' to r'_i
```

<u>Primitive 12.3.</u> **FREE DECOMPOSITION - HIERARCHY GENERATION**

Description:

This primitive generates several relations for the hierarchy. The form (distribution of attributes) of these relations is decided by the designer.

Input:

- source schema : the source schema defined for the group
- $J_1, ..., J_m$, set of attributes that constitutes 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
- $\{ R_{J1}, ..., R_{Jh} \}$, set of relations where the attributes of the hierarchy are distributed / $K \in \mathit{Att}(R_{J1}) \wedge A \in \mathit{Att}(R_{J1})$)
- source instance : $r_1, ..., r_n$

Resulting schema:

- $R_{J1} \in \mathit{Rel}_J \wedge \{K\} \in \mathit{Att}_K(R_{J1})$
- ………………..
- $R_{Jh} \in \mathit{Rel}_J$
- R'_i / $\mathit{Att}(R'_i) = \{ K \} \cup (\mathit{Att}(R_i) - \{ J_1, ..., J_m \}) \wedge \{K\} = \mathit{Att}_{FK}(R'_i, R_{J1})$, i: 1..n

Generated instance:

```
▪ $r_{J1}, ...., r_{Jm}$ =
        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 as corresponds, s into $R_{J1}, R_{J2}, ...., R_{Jh}$
        for each i:1..m / $\forall$ j:1..n, $J_i \notin Att(R_j)$ do
            Fill values of $J_i$ in $R_{J1}, R_{J2}, ...., R_{Jh}$

▪ $r'_i$ = for each tuple t of $r_i$ do
            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 $R_{J1}$
                        where $R_{J1}$.A = t.A
          add t' to $r'_i$
```

Primitive 13. **MINIDIMENSION BREAK OFF**

Description:

This primitive generates two dimension relations. One is the result of eliminating a set of attributes from a dimension relation. The other is a relation that contains only this set of attributes. Besides, it defines a foreign key between the two relations.

Input:

- source schema : R (A_1, ..., A_n) ∈ Rel_D
- K, key for the new dimension
- X ⊂ { A_1, ..., A_n } , set of attributes of the minidimension
- source instance : r

Resulting schema:

- R_1 (A'_1, ..., A'_n) ∈ Rel_D / { A'_1, ..., A'_n } = { A_1, ..., A_n } – X ∪ { K }
- R_2 / $Att(R_2)$ = { K } ∪ X

Generated instance:

Note: For continuously valued attributes such as age or income level, the instance must be pre-processed so that the distinct values of the attributes are grouped into bands.

```
▪ r2 =  select key-gen, X
        from R

▪ r1 =  select R2.K, R.A'1, ...., R.A'n
        from R, R2
        where R.X = R2.X
```

where key-gen is a user-function that must provide the keys for the tuples of R_2.

Example:

CUSTOMERS

NAME	AGE	INCOME_LEVEL	ADDRESS	SEX	CITY	CS
R. Mendez	20	10000	Bvar. Artigas 3	F	Mont.	S
S. Nunez	30	15000	J. Herrera y Ob	M	Mont.	C
M. Garcia	20	10000	Garzon 2125	F	Salto	S
L. Lopez	50	5000	18 de Julio 643	M	Colonia	C

Primitive 13 is applied, where the input is:

- R = CUSTOMERS
- K = DEM_COD
- X = {AGE, INCOME_LEVEL, SEX, CE}
- r = tuples of CUSTOMERS

Result:

DEMOGRAPHICS

DEM_COD	**AGE**	**INCOME_LEVEL**	**SEX**	**CE**
100	20	10000	F	S
200	30	15000	M	C
300	50	5000	M	C

CUSTOMERS

NAME	**ADDRESS**	**CITY**	**DEM_COD**
R. Mendez	Bvar. Artiga	Mont.	100
S. Núñez	J. Herrera y	Mont.	200
M. Garcia	Garzon 2125	Salto	100
L. Lopez	18 de Julio	Colonia	300

♦

Primitive 14. NEW DIMENSION CROSSING

Description:

The source schema is composed of two relations of any type (dimension or crossing), which have an attribute in common. Only one of the relations can contain measure attributes. This primitive generates a crossing relation whose attributes are the union of attribute subsets of the source relations.

Note: If one of the source relations is a measure relation, its relationship with the other source relation must be N:1.

Input:

- source schema : R_1, R_2 / $(R_1, R_2 \in (Rel_D \cup Rel_C) \vee$
  $(R_1 \in Rel_M \wedge R_2 \in (Rel_D \cup Rel_C))) \wedge$
  $Att_K(R_1) = X_1 \wedge Att_K(R_2) = X_2 \wedge$
  $R_1 \cap R_2 = Z$
- Y_1, Y_2, sets of attributes to be excluded from the resulting relation
- N:N, Boolean argument (the relationship between the relations is N:N or not)
- source instance : r_1, r_2

Resulting schema:

```
▪ $R \in Rel_C$ / $Att(R) = \{Att(R_1) - Y_1\} \cup \{Att(R_2) - Y_2\} \wedge$
  if N:N then
      if $R_1, R_2 \in Rel_D$ then
          $Att_K(R) = (X_1 \cup X_2) \wedge$
          $Att_{FK}(R, R_1) = X_1 \wedge Att_{FK}(R, R_2) = X_2$
      else if $R_1, R_2 \in Rel_C$ then
          $Att_K(R) = \cup A / ( A \in (X_1 \cup X_2) \wedge A \in R )$
          $Att_{FK}(R) = \{ W / W \in (Att_{FK}(R_1) \cup Att_{FK}(R_2)) \wedge W \subseteq R \}$
      else if $R_1 \in Rel_C \wedge R_2 \in Rel_D$ then
          $Att_K(R) = ( \cup A / ( A \in X_1 \wedge A \in R ) ) \cup X_2$
          $Att_{FK}(R) = X_2 \cup \{ W / W \in Att_{FK}(R_1) \wedge W \subseteq R \}$
  else   // N:1
      $Att_K(R) = X_1 \wedge$
      if $R_1, R_2 \in Rel_D$ then
          $Att_{FK}(R, R_1) = X_1 \wedge Att_{FK}(R, R_2) = X_2$
      else if $R_1, R_2 \in Rel_C$ then
          $Att_{FK}(R) = \{ W / W \in (Att_{FK}(R_1) \cup Att_{FK}(R_2)) \wedge W \subseteq R \}$
      else if $R_1 \in Rel_C \wedge R_2 \in Rel_D$ then
          $Att_{FK}(R) = X_2 \cup \{ W / W \in Att_{FK}(R_1) \wedge W \subseteq R \}$
```

Generated instance:

```
▪ r =  select distinct {$Att(R_1) - Y_1$} ∪ {$Att(R_2) - Y_2$}
       from $R_1$ $R_2$
       where $R_1.A_1 = R_2.A_1$
```

Example:

ACTIVITIES

STUDENT	**COURSE**
S1	C1
S1	C2
S1	C3
S2	C1
S2	C2
S3	C1
S3	C2
S3	C3

INSTRUCTORS

COURSE	**INSTRUCTOR**
C1	I1
C2	I1
C2	I2
C3	I2

Primitive 14 is applied, where the input is:

- R_1 = ACTIVITIES, R_2 = INSTRUCTORS
- Y_1 = {COURSE}, Y_2 = {COURSE}
- N:N = TRUE
- r_1 = tuples of ACTIVITIES, r_2 = tuples of INSTRUCTORS

Result:

STUDENT-INSTRUCTOR

STUDENT	**INSTRUCTOR**
S1	I1
S1	I2
S2	I1
S2	I2
S3	I1
S3	I2

♦

5. Consistency Rules

These are some rules that should be applied always, when a DW schema is constructed through application of the primitives. The goal of these rules is to assure that the obtained DW schema is consistent. We consider a DW schema consistent when it satisfies the DW schema invariants (defined in Section 3).

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, R2 and R3 correspond to the case of invariants I1, I4 and I3 violation, respectively.

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 $R \in Rel_M$, removing $A \in Att_D(R)$, obtaining relation R'

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

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

R3 – History relations update

R3.1 –

ON APPLICATION OF: *Data Filter* to $R_1 \in Rel_H(R)$, removing $A \in Att_{FK}(R_1,R)$, obtaining R_2

APPLY: *Foreign Key Update* to R_2, obtaining R_3, where $A \in Att(R_3) \wedge A \in Att_{FK}(R_3, R)$

R3.2[4] **–**

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

WHEN: $\exists\, R'$ / $R' \in Rel_H(R)$

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

[4] This rule is optional. The user chooses if the rule is active or not.

6. Design Strategies

Strategies for application of primitives are designed taking 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 relationships between dimensions, data summarisation and data crossing, hierarchies' management, and derived data. We select these problems basing on the literature [Kim96-1][Kim96-3][Sil97] and on our own experience.

1. 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 cases, 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:

S1) Apply **Temporalization** primitive (P3), such that the time attribute belongs to the key of the relation.

S2) Generalise 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.

S3) Add new attributes, so that a small number of versions of certain data can be maintained. Do this, applying the primitive **Attribute Adding** (P7).

S4) Generalise 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).

S5) Partition the relation according to its stability through one of the primitives of **Partition by Stability** family (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 **S1**, **S2** or **S4** to the history relation generated.

Example:

CUSTOMERS

SSN	**NAME**	**AGE**	**INCOME**	**ADDRESS**	**SEX**	**CITY**	**CS**
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

2 different options

CUSTOMERS_1

GR_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	....

♦

2. VERSIONING OF N:1 RELATIONSHIPS BETWEEN DIMENSIONS

Frequently, it is necessary to maintain the history about the relationships between the elements of two dimensions. In particular, we will treat the case where originally we have a dimension relation that has a N:1 relationship with another dimension relation, and is referenced from a measure relation. They are connected through foreign keys. In order to be able to maintain the history of the dimensions' relationship, some transformations in the schema has to be applied.

First of all, the designer has to make some decisions:

a) Which is the history he really wants to maintain and how he wants to do it

1- Maintain the history only in the dimension.

In this case the complete history of the relationship with the other dimension will be maintained, and it will be accessible from the dimension.

2- Maintain the history through the data recorded in the measure relation that references the dimension.
Here, it may happen that some states of the relationship between the dimensions are not recorded. Besides, the way to obtain information about the history of the relationships of a dimension's subject, is not direct.

b) Which is the desired design style

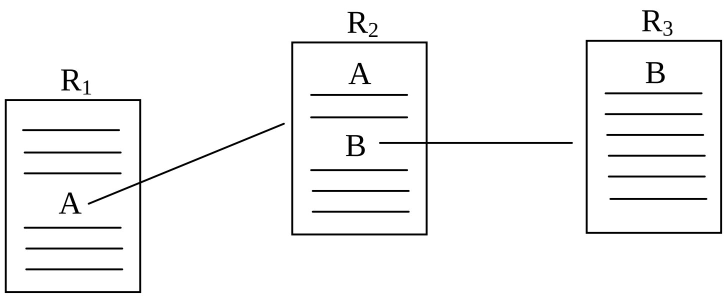
1- Normalised

2- De-normalised

Here we propose four different strategies that can be followed to obtain the desired design. There is a suitable strategy for each of the possible decisions made by the designer. In the following table we show the strategy that must be applied for each combination of type of history and type of design chosen.

design		**history**	
		a-1	a-2
	b-1	S 1	S 2
	b-2	S 3	S 4

Possible strategies:

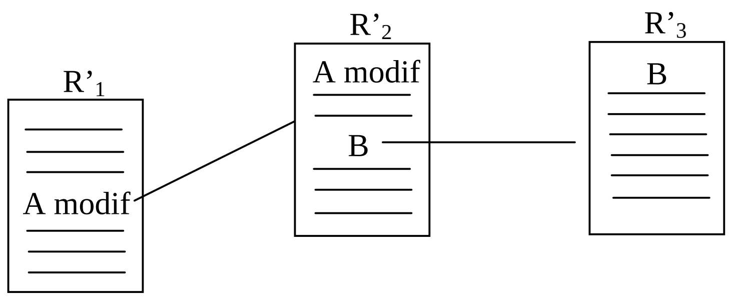


Given a measure relation R_1 and two dimension relations R_2 and R_3, where there is a N:1 relationship between R_1 and R_2 and a N:1 relationship between R_2 and R_3, which slowly changes[5], the possible applicable strategies are the following:

S1) Do a versioning of relation R_2. Due to the consistency rule R_1, it also will be necessary to update relation R_1 so that it references to R_2.

[5] "slowly change" is an expression used by R. Kimball [Kim96-1] referring to data that evolve slowly.

The obtained schema will allow storing several tuples corresponding to the same element of relation R_2, so that each one can reference to a different element of R_3.



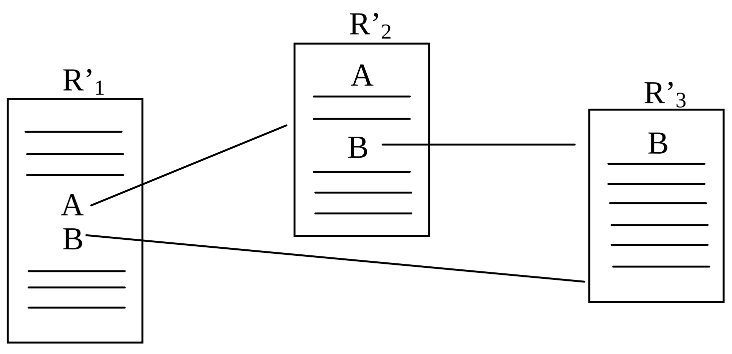
Steps:

1) Apply to R_2 alternatives **S1)**, **S2)** or **S4)** of the Versioning strategies presented earlier.

2) Apply R1 consistency rule.

S2) Modify the measure relation R_1 so that, in addition to referencing relation R_2, it references relation R_3.

With the obtained schema, each movement of the measure relation will reference to an element of R_2 and to an element of R_3, and each element of R_2 will reference to only one of R_3. The idea is that the elements of R_2 reference only to the current corresponding element of R_3.

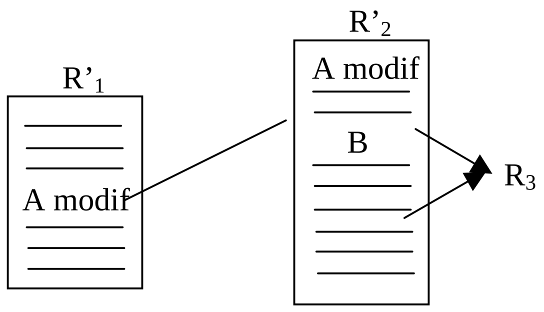


Steps:

1) Apply to R_1 Primitive **DD-Adding n-1** (P6.2), adding to R_1 the attribute that is key of R_3. Derive this attribute from R_2 and declare it as foreign key to R_3.

S3) Do a versioning of relation R_2. Due to the consistency rule R_1, it also will be necessary to update relation R_1 so that it references to R_2. Afterwards, include the attributes of R_3 in R_2 (de-normalising).

The obtained schema will allow storing several tuples corresponding to the same element of relation R_2, but containing different data obtained from relation R_3.



Steps:

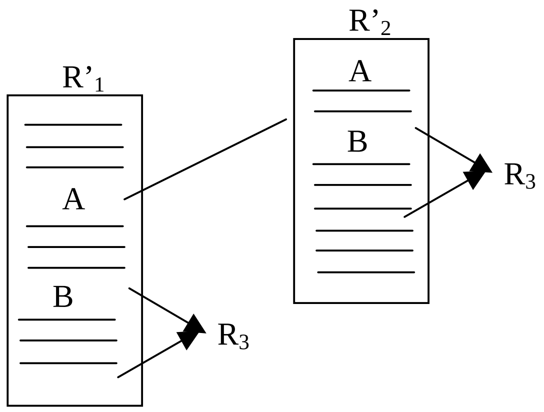
1) Apply to R_2 the alternatives **S1)**, **S2)** or **S4)** of the Versioning strategies presented earlier.

2) Apply R1 consistency rule.

3) Apply to R_2 Primitive **DD-Adding n-1** (P6.2), successively, adding the desired attributes of R_3. Derive these attributes from R_3.

S4) Include the attributes of relation R_3 in relation R_1 and in relation R_2 (de-normalising).

With the obtained schema, each movement of the measure relation will reference to an element of R_2 and will contain the corresponding data of R_3, and each element of R_2 will contain the data of only one of R_3. The idea is that the elements of R_2 contain only the data of the current corresponding element of R_3.



Steps:

1) Apply to R_2 Primitive **DD-Adding n-1** (P6.2), successively, adding the desired attributes of R_3. Derive these attributes from R_3.

2) Apply to R1 Primitive **DD-Adding n-1** (P6.2), successively, adding the desired attributes of R_3. Derive these attributes from R_2.

3. AGGREGATES AND DATA CROSSINGS

As a consequence of the type of requirements that in general exist over a DW, there is a large number of different data crossings and different level of summarisations that should be materialised in the DW. Therefore, measure and crossing relations are the most common type of relations that are constructed during a DW design.

The new crossing relations are constructed from existing relations that use to be dimension, hierarchy and crossing relations.

The following are some general cases that may appear in this context, and the existing alternatives for constructing the new relations through application of the primitives.

S1) There is a measure relation where one of the attributes is part of a hierarchy that exists in another relation. It is required to increase the level of this attribute in the measure relation, following the hierarchy.

Two options exist for the generated sub-schema:

1.2) A new measure relation equal to the original one, except for one of its attributes, which corresponds to a higher level in the hierarchy. The data will be at the same or higher summarisation level.

1.3) The same measure relation as in **1.2)** and in addition, a new hierarchy relation where the lower level is the same as the level chosen for the attribute of the measure relation.

For obtaining any of these two results, apply Primitive **Hierarchy Roll Up** (P8), specifying in the input if a new hierarchy relation is wanted or not.

S2) Given a measure relation, the designer wants to group information by some of the attributes of the relation.

In this case a new measure relation is constructed. In this relation data will be grouped by some of the attributes of the original relation. The attributes included in the new relation are only the ones that correspond to the new grain. For obtaining this result apply Primitive **Aggregate Generation** (P9).

S3) It is required to obtain new data combinations structured in crossing relations, starting from different types of relations.

The relations to be combined may be of dimension or crossing type, and only one of them can be a measure relation. These relations must have some attributes in common so that they can be joined. The new crossing relation will have attributes of the two original relations, filtering the attributes of no interest for the new crossing. For obtaining this result apply Primitive **New Dimension Crossing** (P14).

S4) Combinations of the cases above.

Compose Primitives **Hierarchy Roll Up**, **Aggregate Generation** and **New Dimension Crossing** (P8, P9 and P14).

Examples:

a) We want to construct a crossing relation combining data from a measure and a crossing relation. Then we want to group by some attributes of the new relation.

PROD-MONTH

PRODUCT	**MONTH**	**QUANTITY**
P1	5	100
P1	6	120
P2	5	50
P3	5	300

PROD-SUP

PRODUCT	**SUPPLIER**
P1	PR1
P2	PR1
P3	PR3
P4	PR4

Crossing

PROD-SUP-MONTH

PRODUCT	**SUPPLIER**	**MONTH**	**QUANTITY**
P1	PR1	5	100
P1	PR1	6	120
P2	PR1	5	50
P3	PR3	5	30

Group by Supplier and Month

SUP-MONTH

SUPPLIER	**MONTH**	**QUANTITY**
PR1	5	150
PR1	6	120
PR3	5	30

♦

b) We want to construct a measure relation that combines data from two crossing relations. In this case a new measure attribute is generated.

ACTIVITIES

STUDENT	**COURSE**
E1	C1
E1	C2
E1	C3
E2	C1
E2	C2
E3	C1
E3	C2
E3	C3

INSTRUCTOR

COURSE	**INSTRUCTOR**
C1	D1
C2	D1
C3	D2

Crossing

STUDENT	**COURSE**	**INSTRUCTOR**
E1	C1	D1
E1	C2	D1
E1	C3	D2
E2	C1	D1
E2	C2	D1
E3	C1	D1
E3	C2	D1
E3	C3	D2

Group by Instructor

INSTRUCTOR	**STUD_QUANT**
D1	5
D2	2

♦

4. IDENTIFICATION AND SEPARATION OF HIERARCHIES

Frequently, in operational databases we can find embedded in relations sets of attributes where exists a hierarchy relation between the attributes. Besides, in cases where the database is obtained from several different sources, it may happen that the same hierarchy is repeated in different representations.

In general, with respect to a relation that "includes" a hierarchy we can find two different situations:

1) All the attributes of the hierarchy belong to the relation.

2) The relation has an attribute of the hierarchy that references to the rest of the hierarchy, which can be distributed in several relations.

A reasonable possibility in a DW schema is that a set of attributes that semantically constitute a hierarchy exists in the schema only once and is reused by all the relations that need to reference to it. This also allows that relations that contain a subset of the considered hierarchy, can reference the whole hierarchy and therefore can make new groupings of its data.

In order to perform a reorganisation and cleaning of all relative to the hierarchies in a schema, we propose to follow these steps:

1) Select all the relations of the schema that include a hierarchy or part of one.

2) With the selected relations, form groups of relations that correspond to the same hierarchy.

3) For each group do:

   a) For each relation that references a hierarchy that is in other relations (situation 2)) do:
      Apply Primitive **New Dimension Crossing** (P14) to all involved relations, obtaining only one relation.

   b) Determine the attributes of the hierarchy to be constructed and its key.

   c) Apply Primitive **Hierarchy Generation** (P12) to all relations of the group. In this step there are three possible design alternatives with respect to the hierarchy to be constructed:

      i) De-normalised. All the attributes of the hierarchy belongs to the same relation. (P12.1)

      ii) Normalised. The attributes of the hierarchy are distributed in several relations, each one containing two attributes. (P12.2)

      iii) Distributed in several relations according to some designer's criteria. (P12.3)

Example:

Suppose we have already done steps **1)** and **2),** and one of the groups of relations we obtained is composed by relations Branches, Customers, Suppliers and Supp-Location.

BRANCHES

BRAN_CODE	**BRAN_NAME**	**ADDRESS**	**MANAGER**	**CITY**	**COUNTRY**
C1	A	Bvar. Artiga	Juan Perez	Montevideo	Uruguay
C2	B	J. Herrera y	Pepe Diaz	Montevideo	Uruguay
C3	C	19 de Junio	Maria Suarez	Bs. As.	Argentina
C4	D	Calle A 334	Jose Sanchez	Bs. As.	Argentina

CUSTOMERS

CUST_CODE	**CUST_NAME**	**ADDRESS**	**CITY**	**REGION**	**COUNTRY**
C1	Empresa ABC	18 de Julio 1	Montevideo	Montevideo	Uruguay
C2	Ramirez Hnos.	Rambla Arm	Montevideo	Montevideo	Uruguay
C3	Daniel Kual	19 de Junio	Bs. As.	Bs. As.	Argentina
C4	Nuvoses	Calle de los	La Plata	Bs. As.	Argentina

SUPPLIERS (CITY foreign key to SUPP-LOCATION)

SUPP_CODE	**SUPP_NAME**	**ADDRESS**	**CITY**
S1	AAAA	Bvar. Artiga	Montevideo
S2	BBBB	J. Herrera y	Montevideo
S3	CCCC	19 de Junio	Bs. As.
S4	DDDD	Calle A 334	La Plata

SUPP-LOCATION

CITY	**REGION**	**COUNTRY**
Montevideo	Montevideo	Uruguay
Bs. As.	Bs. As.	Argentina
La Plata	Bs. As.	Argentina

Now we will perform step **3)**. First we apply **a)** to relations Suppliers and Supp-Location and we obtain a new relation Suppliers.

SUPPLIERS

SUPP_CODE	**SUPP_NAME**	**ADDRESS**	**CITY**	**REGION**	**COUNTRY**
S1	AAAA	Bvar. Artiga	Montevideo	Montevideo	Uruguay
S2	BBBB	J. Herrera y	Montevideo	Montevideo	Uruguay
S3	CCCC	19 de Junio	Bs. As.	Bs. As.	Argentina
S4	DDDD	Calle A 334	La Plata	Bs. As.	Argentina

According to **b)** we have to determine the hierarchy we want to construct.

Hierarchy's attributes: city, region, country Key: geo_cod

Following step **c)**, we apply Primitive 12.1 generating new Branch, Customers and Suppliers relations and a de-normalised hierarchy Geography.

BRANCHES

BRAN_CODE	**BRAN_NAME**	**ADDRESS**	**MANAGER**	**GEO_COD**
C1	A	Bvar. Artiga	Juan Perez	G01
C2	B	J. Herrera y	Pepe Diaz	G01
C3	C	19 de Junio	Maria Suarez	G02
C4	D	Calle A 334	Jose Sanchez	G02

CUSTOMERS

CUST_CODE	**CUST_NAME**	**ADDRESS**	**GEO_COD**
C1	Empresa ABC	18 de Julio 1	G01
C2	Ramirez Hnos.	Rambla Arm	G01
C3	Daniel Kual	19 de Junio	G02
C4	Nuvoses	Calle de los	G03

SUPPLIERS

SUPP_CODE	**SUPP_NAME**	**ADDRESS**	**GEO_COD**
S1	AAAA	Bvar. Artiga	G01
S2	BBBB	J. Herrera y	G01
S3	CCCC	19 de Junio	G02
S4	DDDD	Calle A 334	G03

GEOGRAPHY

GEO_COD	**CITY**	**REGION**	**COUNTRY**
G01	Montevideo	Montevideo	Uruguay
G02	Bs. As.	Bs. As.	Argentina
G03	La Plata	Bs. As.	Argentina

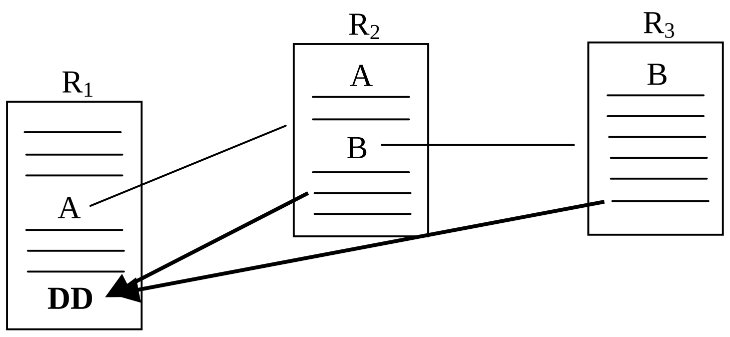
♦

5. DERIVED DATA

In general, in a DW is useful to have attributes whose value is derived from others, which can be stored in other relations, in order to simplify and accelerate queries.

When it is necessary to add to a relation R_1 an attribute that is calculated from other relations R_2, ..., R_n, one of the following situations may happen:

S1) Each value of the attribute is calculated from values of attributes that belong to only one tuple obtained from the R_2, ..., R_n join.



The steps to follow in order to generate the derived attribute in R_1 are the following:

a) If n>2 then

Apply Primitive **New Dimension Crossing** (P14) to relations R_2, ..., R_n, obtaining R'.

b) If n=2 then

Apply Primitive **DD-Adding n-1** (P6.2) to R_1 and R_2.

Else if n>2 then

Apply Primitive **DD-Adding n-1** (P6.2) to R_1 and R'.

Example:

CUSTOMERS

SSN	**NAME**	**ADDRESS**	**PHONE**	**PLAN**	**QUOTE**	**CURR_QUOTE**
2760527	Juan Perez	B. Artigas 444	121212	100	2	490
5321532	Maria Lopez	G. Flores 2255	545454	101	1	315

PLANS

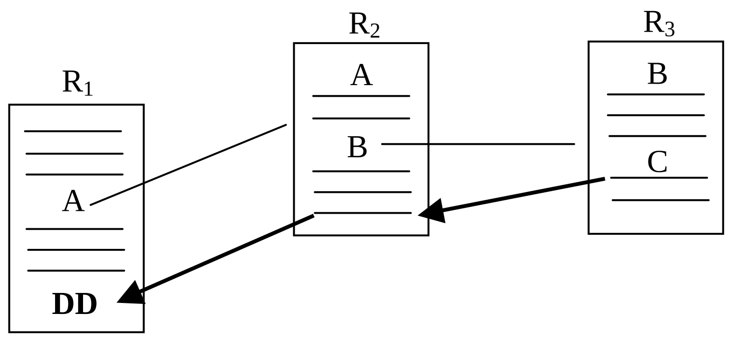
PLAN	**QUOTE**	**QUOTE_VALUE**
100	1	500
100	2	495
100	3	490
101	1	350

DISCOUNTS

PLAN	**DISC%**
100	1
101	10
102	7
103	3

♦

S2) Each value of the attribute is calculated from the composition of the aggregations of values of the attributes belonging to the relations R_2, ..., R_n.



The steps to follow in order to generate the derived attribute in R_1 are the following:

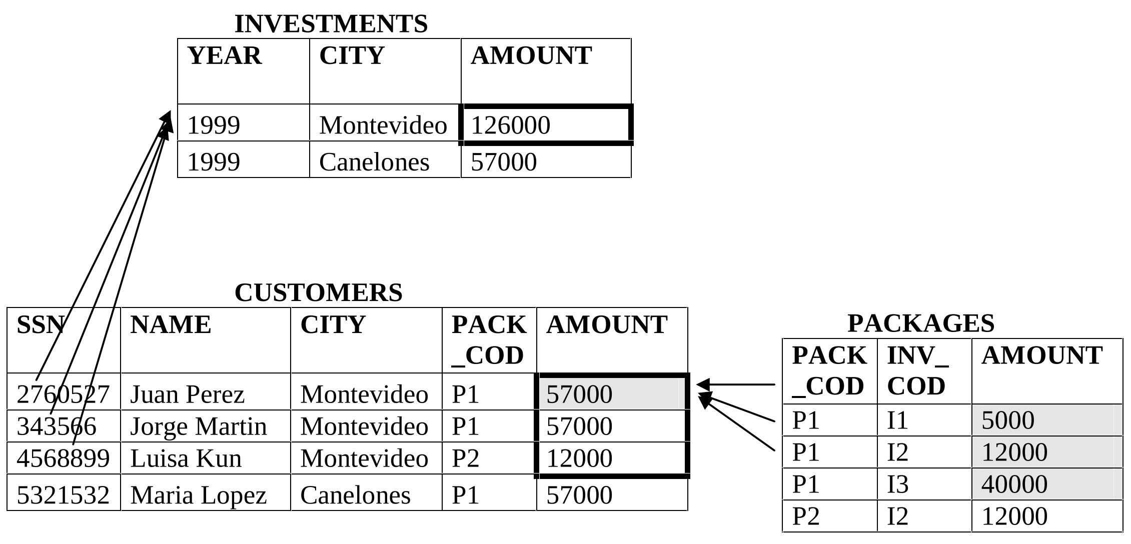
a) If n=2 then

Apply Primitive **DD-Adding n-n** (P6.3) to relations R_1 and R_2.

Else if n>2 then

Compose applications of Primitive **DD-Adding n-n**, starting from the two relations with highest grain and then applying the primitive successively to the last result and the following highest grain relation.

Example:



INVESTMENTS

YEAR	CITY	AMOUNT
1999	Montevideo	126000
1999	Canelones	57000

CUSTOMERS

SSN	NAME	CITY	PACK _COD	AMOUNT
2760527	Juan Perez	Montevideo	P1	57000
343566	Jorge Martin	Montevideo	P1	57000
4568899	Luisa Kun	Montevideo	P2	12000
5321532	Maria Lopez	Canelones	P1	57000

PACKAGES

PACK _COD	INV_ COD	AMOUNT
P1	I1	5000
P1	I2	12000
P1	I3	40000
P2	I2	12000

♦

7. Transformation trace

In this section we present how we manage and specify the trace of the transformation that was applied to a source database schema in order to obtain a DW schema.

In our proposal DW design is a subsequent application of primitives in a composition mode. The result of this application is a schema where each relation is obtained by application of primitives.

In most cases a final sub-schema is not obtained through application of one primitive to a sub-schema of the source schema, but it is obtained through composition of several primitives. We call this process a *sequence of primitive applications*.

The subsequent primitive application generates a trace of the transformation made. Therefore, for each element of the final schema there is a trace that can be seen as the path that was followed for obtaining this element starting from a source element. This trace provides the information about the sequences of primitives that were applied to the source element.

In the following section we give a way to represent and specify the trace of a schema design.

7.1.1. Trace specification

We specify the trace of a schema design using a set of expressions with the form of function applications.

By means of this specification we can use the trace starting from elements of the final schema in order to know their origin. We obtain a mapping that is necessary for the construction of the processes for loading data from the source database to the constructed DW.

At the same time this specification allows us to use the trace starting from elements of the source schema. This perspective is necessary for propagating changes that these elements have suffered to the DW schema.

Definition: Transformation Trace ***T***

Given a set of relations, a set of attributes, a set of functions and a set of primitives, the Transformation Trace is represented by the following grammar:

T ::= <exp_set>

<exp_set> ::= <rel_set> ‘=’ <prim_app> | <rel_set> ‘=’ <prim_app> ‘;’ <exp_set>

<rel_set> ::= ‘{’ <relations> ‘}’ | <relation>

<relations> ::= <relation> | <relation> ‘,’ <relations>

<relation> ::= *Rel_Name*

<prim_app> ::= <primitive> ‘(’ <rel_set> ‘,’ <arg_list> ‘)’ |
<primitive> ‘(’ <prim_app> ‘,’ <arg_list> ‘)’

<primitive> ::= *Primitive_Name*

<arg_list> ::= <argument> | <argument> ‘,’ <arg_list>

<argument> ::= <rel_set> | <att_set> | <function_set> | *Boolean* | ∅

<att_set> ::= ‘{’ <attributes> ‘}’ | <attribute>

```
<attributes> ::=  <attribute> | <attribute> ‘,’ <attributes>
<attribute> ::=  Att_Name
<function_set> ::= ‘{’ <functions> ‘}’ | <function>
<functions> ::=  <function> | <function> ‘,’ <functions>
<function> ::=  Fun_Name
```

Note that this grammar does not control the validity of the arguments (quantity and types) passed to each primitive. We complement it with the following restriction expressed in natural language:

> The <prim_app> expression must respect the format of the input of the primitive, which is stated in the specification of the primitive.

In a concrete application these expressions are complemented with the specifications of the relations.

♦

Example: The representation of part of a schema design trace.

{TIME_MONTH, MONTH_SALES} = **P8** ({SALES, TIME}, {quantity}, month, {sum(quantity)}, ∅, {date, week}, true)

CMP_SALES = **P9** (MONTH_SALES, {quantity_m}, {sum(quantity_m)}, {salesman, city})

{CUSTOMERS_1, DEMOGRAPHICS} = **P13** (CUSTOMERS, dem_code, {age, income_level, sex, ce})

CUSTOMERS_DW = **P3** (CUSTOMERS_1, date, true)

Relation schemas:

SALES (customer, salesman, date, prod, city, quantity)
TIME (date, week, month, trimester, year)
CUSTOMERS (name, age, income_level, address, sex, city, cs)
MONTH_SALES (customer, salesman, month, prod, city, quantity_m)
CUSTOMERS_1 (name, address, city, dem_code)
CMP_SALES (customer, month, prod, quantity_cmp)
TIME_MONTH (month, trimester, year)
CUSTOMERS_DW (name, date, address, city, dem_code)
DEMOGRAPHICS (dem_cod, age, income_level, ce)

♦

In addition we use a graphic representation, a directed acyclic graph *G(T)*, which main goal is to show a global perspective of the process. This representation facilitates the comprehension and localisation of the trace of a certain element.

We complement this graph with textual representation of: (i) the structure of the relations, and (ii) the input arguments of each primitive application. We do not include these specifications in the graph for readability reasons.

Definition: Graph ***G(T)***.

G(T) is a directed acyclic graph composed by the following:

Nodes: Three types of nodes.

1) ◯ - Represents the application of a primitive.

2) □ - Represents a relation.

3) ⬭ - Represents a list of external arguments for a primitive.

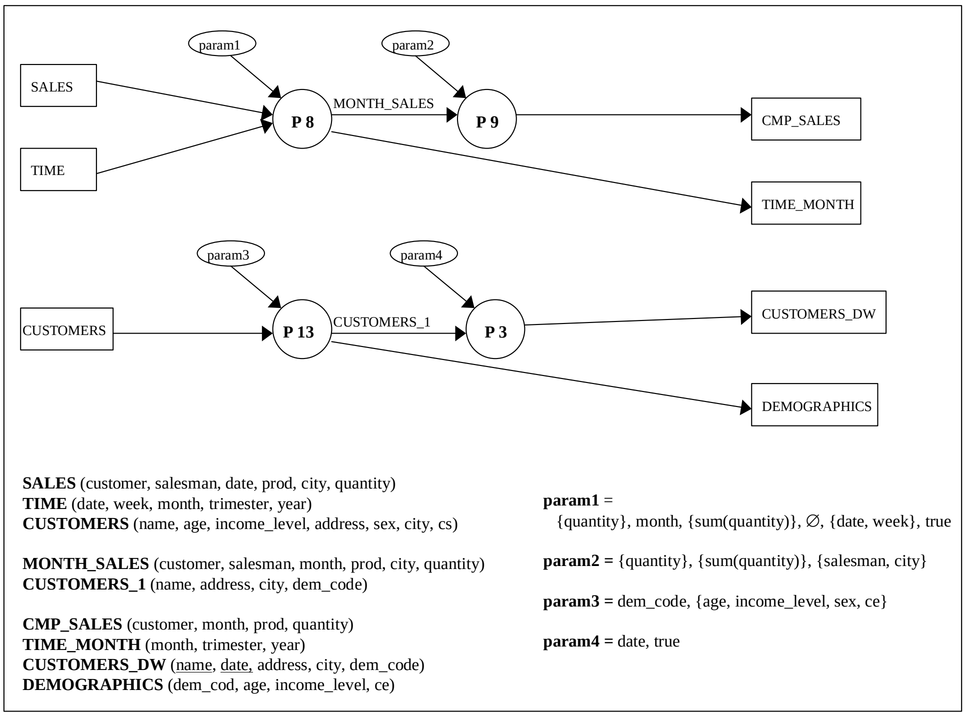
Edges:

- Each edge joins : (a) a relation with a primitive, (b) two primitives, (c) a primitive with a relation, or (d) a list of arguments with a primitive. The representations in each case are the following: in (a) the relation is part of the input of the primitive, in (b) part of the output of one primitive is the input of the other one, in (c) the relation is part of the output of the primitive, and in (d) the arguments are part of the input of the primitive.

- The edges are labelled when necessary. (Edges need to be labelled only when they are joining two primitives). The label of an edge is the name of a relation

♦

Example: Figure 3.3 shows the graph corresponding to the trace specified in the previous example.



SALES (customer, salesman, date, prod, city, quantity)
TIME (date, week, month, trimester, year)
CUSTOMERS (name, age, income_level, address, sex, city, cs)

MONTH_SALES (customer, salesman, month, prod, city, quantity)
CUSTOMERS_1 (name, address, city, dem_code)

CMP_SALES (customer, month, prod, quantity)
TIME_MONTH (month, trimester, year)
CUSTOMERS_DW (name, date, address, city, dem_code)
DEMOGRAPHICS (dem_cod, age, income_level, ce)

param1 =
{quantity}, month, {sum(quantity)}, ∅, {date, week}, true

param2 = {quantity}, {sum(quantity)}, {salesman, city}

param3 = dem_code, {age, income_level, sex, ce}

param4 = date, true

Figure 3.3 Graph representing a trace

♦

8. Conclusion

This paper addresses the problem of DW design, presenting a framework for designing DWs by application of schema transformations

We present a help tool for DW design, which is a set of schema transformation primitives complemented with some strategies and rules for their practical application. These transformation primitives enable to design a relational DW from a source relational schema, acting as design building blocks that have DW design knowledge embedded in their semantics. In addition, the application of these primitives provides a trace, which will be the trace of the design. Utilisation of design building-blocks improves quality and productivity in the design. On the other hand, the design trace is an important tool for documentation and design process management, and it is essential for performing DW maintenance. In particular, it enables to perform the repercussion of source schema evolution to the DW.

In our proposal 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.

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.

A proposal about the automatic application of the primitives, starting from the conceptual model (high level vision about the information requirements) and the correspondences with the source schemas, is being developed as part of a master thesis [Per00].

We are also working on the problem of propagating source schema evolution to the DW taking advantage of the trace generated during the design.

In the future the following additional issues could be addressed:

- experimentation with the primitives in different applications and generation of new versions of the set of primitives

  We believe that the set of primitives can be improved in some ways. Experimentation with it shows that correcting some parameters of some of the primitives, their application would be more flexible and simpler.

- inclusion of schema integration facilities to the primitives

  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.

- completeness of the primitives

  Primitive completeness could be informally shown by testing them in a wide gamma of scenarios, applying different techniques, in different application areas, etc.

Another way to show it, is trying to apply the different design proposals that can be found in the bibliography, through the primitives.

- data loading and maintenance

  Together with each primitive, we provide an outline of the transformation that should be done to the existing data for populating the generated sub-schema. For solving the problem of data loading and maintenance much more work must be done in this direction.

- application to real cases of the proposed mechanism for managing evolution

  It would be interesting to apply the proposed mechanism for source schema evolution to real cases, as we did with the primitives.

- evolution generated by changes in DW requirements

  We are working on a solution for DW schema evolution that was generated by evolution of the source schemas. DW schema evolution generated by changes in DW requirements is an important problem, which we have not addressed yet.

References

[Gut00] A. Gutiérrez, A. Marotta. An Overview of Data Warehouse Design Techniques. Reporte Técnico INCO-01-09. InCo - Pedeciba, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay. Noviembre 2000. ISSN 0797-6410.

[Kim96-1] R. Kimball. *The Data Warehouse Toolkit.* J. Wiley & Sons, Inc. 1996

[Kim96-3] R. Kimball. *Slowly Changing Dimensions.* The Data Warehouse Architect, DBMS Magazine, April 1996, URL: http://www.dbmsmag.com

[Per00] V. Peralta *Sobre el pasaje del esquema conceptual al esquema lógico de un Data Warehouse.* Facultad de Ingenieria. Universidad de la República del Uruguay. In.Co. Reporte Interno. 2000.

[Sil97] L. Silverston, W. H. Inmon, K. Graziano. *The Data Model Resource Book.* J. Wiley & Sons, Inc. 1997

Appendix – An Application Example

This is a case of a product distribution company who wants to construct a DW. The most important requirements are related to: (i) sales evolution by product families and geographic regions, (ii) product cost analysis, (iii) market analysis (types of clients), and (iv) geographic distribution of the sales.

The source database schema is shown in **Figure 7**, which is a representation of the relational schema, where the lines represent the links between the tables through the foreign keys.

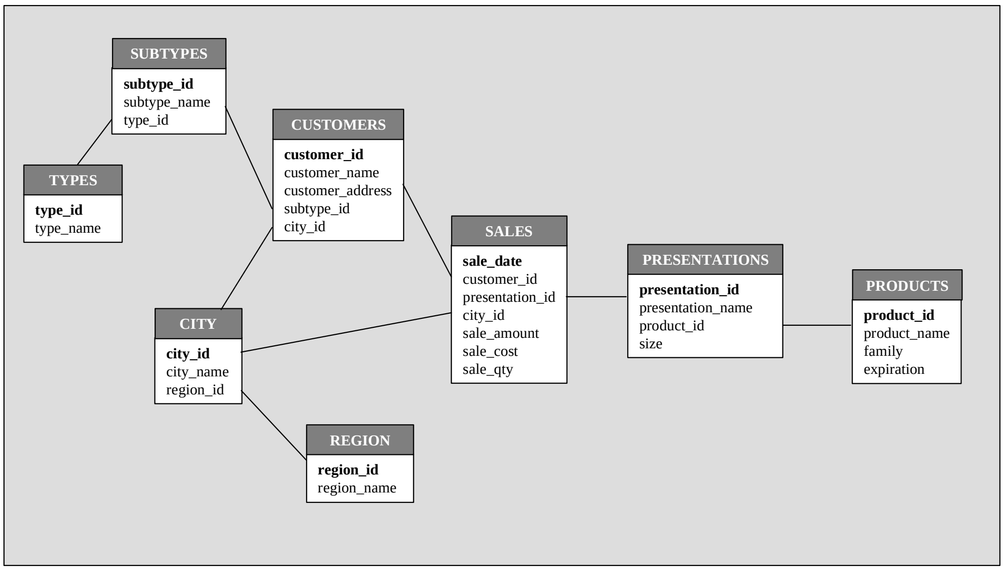


Figure 7: The source database schema

We suppose that, following one of the existing DW design methodologies [Kim96-1][Kor99][Bal98], we arrived to the design presented in **Figure 8**. It is a star schema[6], where the dimensions are Time, Customers_DW, Products_DW, and Geography, and the fact table is Sales_DW, where sale_amount, sale_cost and sale_qty are the measures.

[6] Star Schema is defined in [Kim96-1]

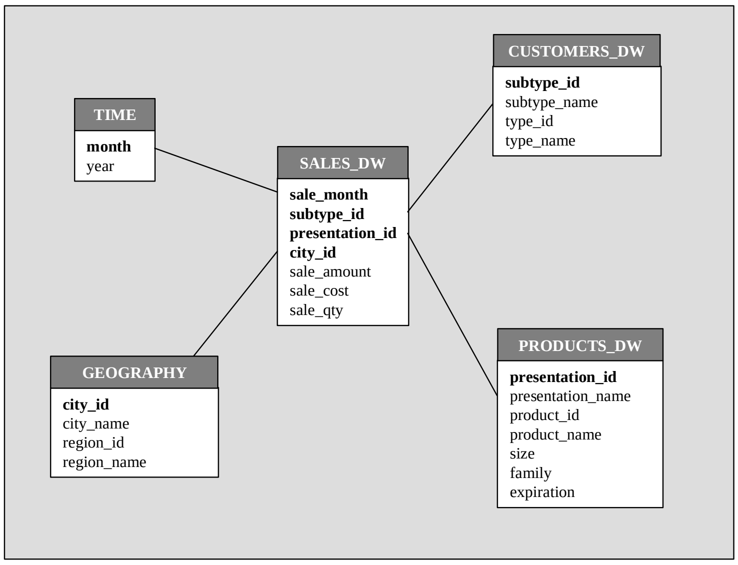


Figure 8: The target logical DW schema

Now, we apply the transformation primitives to the source schema in order to generate the desired DW schema.

First, we de-normalise the relations that correspond to the dimensions, generating a new relation for each dimension of the desired schema. We use primitive **P6.2 DD-Adding 1-N** for adding the attributes from one relation to the other relation.

Products_DW: We apply **P6.2** to relations *Presentations* and *Products*, obtaining:

PRODUCTS_DW (**presentation_id**, presentation_name, product_id, product_name, size, family, expiration)

Customers_DW: We apply **P6.2** to relations *Customers*, *Subtypes* and *Types*, obtaining:

CUSTOMERS_DW_01 (**customer_id**, customer_name, customer_address, subtype_id, city_id, subtype_name, type_id, type_name)

Geography: We apply **P6.2** to relations *City* and *Region*, obtaining:

GEOGRAPHY (**city_id**, city_name, region_id, region_name)

CUSTOMERS_DW_01 has some attributes that are not relevant for this case. We apply primitive **P2 Data Filter** for eliminating them.

Customers_DW: We apply **P2** to relation *Customers_DW_01*, obtaining:

CUSTOMERS_DW_02 (**customer_id**, subtype_id, subtype_name, type_id, type_name)

For the *Time* dimension we obtain the *date* attribute from the *Sales* relation. We do this through the primitive **P12.1 De-Normalized Hierarchy Generation**, which generates a hierarchy relation from relations that contain a whole hierarchy or a part of one. Then we calculate the attributes *month* and *year* from the *date*, using primitive **P6.1 DD_Adding 1-1**.

Time: We apply **P12.1** to *Sales*, obtaining:

TIME_01 (**date**)

and we apply twice **P6.1** to TIME_01 for adding attributes month and year:

TIME_02 (**date**, month)

TIME_03 (**date**, month, year)

For generating the fact table (measure relation) *Sales* with the desired granularity, which is *subtype* for *Customer* dimension and *month* for *Time* dimension, we apply the primitive **P8 Hierarchy Roll-Up**. This primitive also changes the level of detail of the dimensions. The summarisation function for each measure must be specified to the primitive. In this case it is the sum function.

Sales_DW: We apply **P8** to *Sales* and *Customers_DW_02*, obtaining:

SALES_DW_01 (**sale_date**, **subtype_id**, **presentation_id**, **city_id**, sale_amount, sale_cost, sale_qty)

and

CUSTOMERS_DW (**subtype_id**, subtype_name, type_id, type_name)

We apply **P8** to Sales_DW_01 and Time_03, obtaining:

SALES_DW (**sale_month**, **subtype_id**, **presentation_id**, **city_id**, sale_amount, sale_cost, sale_qty)

and

TIME (**month**, year)

Through the applied primitives we generated the desired schema, showed in **Figure 8**.

Now we will refine the design. Suppose we detect that the Product dimension has some attributes (size, family) that change their values through time. According to definitions in [Kim96][Kim97] it is a *slowly changing dimension*. We decide that, for query performance reasons, we will maintain this history data in a separate relation. For this, we follow two steps. First, we apply **P11.2 Horizontal Partition** to *Products_DW* relation for generating a new relation for the history data. Second, we apply **P3 Temporalization** to the history relation adding the time attribute to the key of the relation.

Products_DW_His: We apply **P11.2** to *Products_DW*, obtaining:

PRODUCTS_DW_HIS_01 (**presentation_id**, presentation_name, product_id, product_name, size, family, expiration)

We apply T3 to *Products_DW_His_01*, obtaining:

PRODUCTS_DW_HIS (**presentation_id**, **change_date**, presentation_name, product_id, product_name, size, family, expiration)

Finally, also for performance reasons, we want to add to *Geography* relation a calculated attribute *cust_qty*, which represents the quantity of customers that belongs to each city. We do this through the application of the primitive **P6.3 DD_Adding N-N**, which adds to a relation an attribute that is calculated from the summarisation of many tuples of other relation.

Geography_Cust: We apply T6.3 to *Geography* and *Customers*, obtaining:

GEOGRAPHY_CUST (**city_id**, city_name, region_id, region_name, cust_qty)

The final DW schema is shown in **Figure 9**.

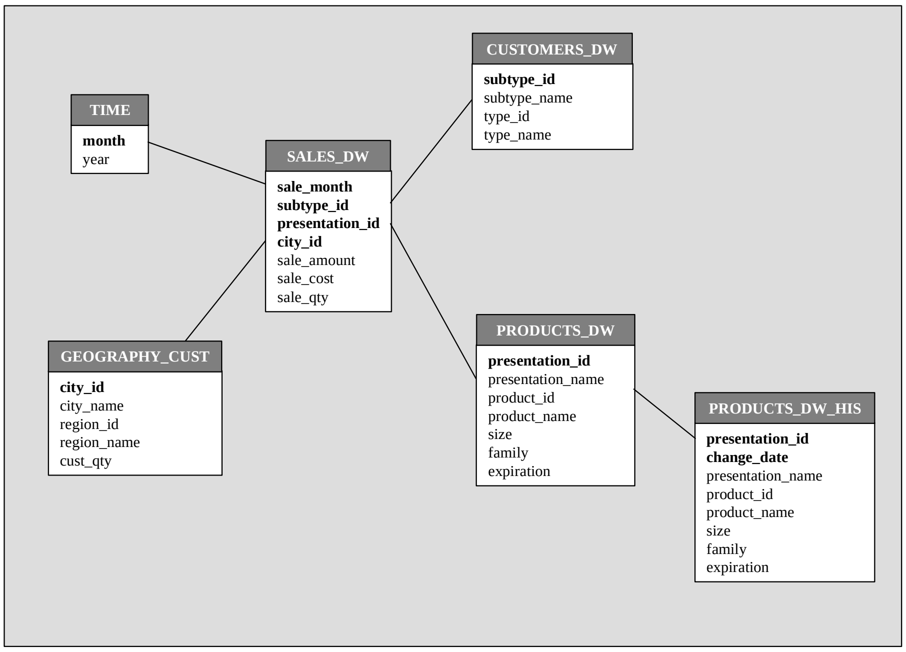


Figure 9: The obtained DW schema

The applied primitives generate a trace of the design, which is shown in **Figure 10**.

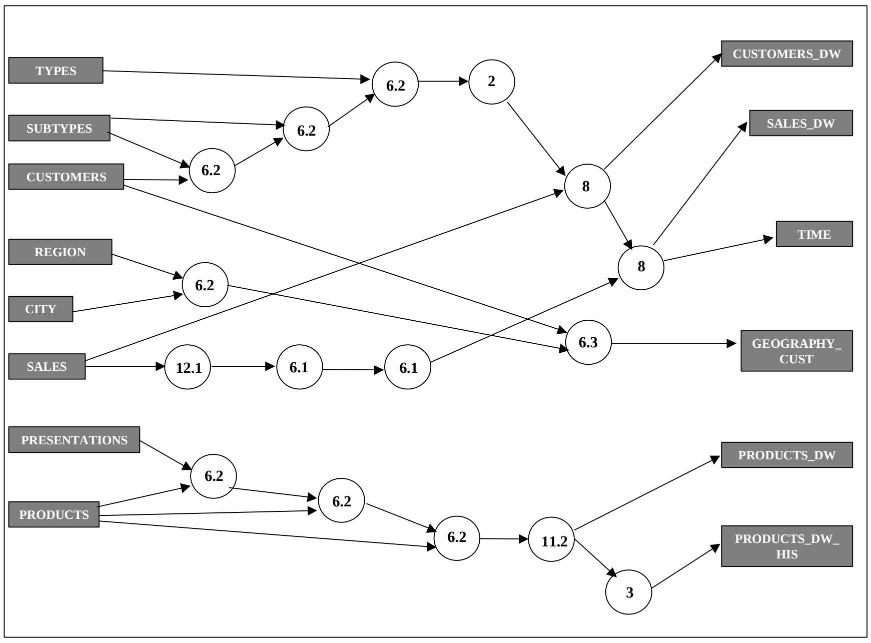


Figure 10: The generated trace