Managing source schema evolution in relational data warehouses [‡]

DRAFT

Adriana Marotta

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

December 2001

Abstract

Schema evolution in a DW can be generated by two different causes: (i) a change in the source schema or (ii) a change in the DW requirements. In this work we address the problem of source schema evolution. We separate this problem into two phases: (1) determination of the changes that must be done to the DW schema and to the trace, and (2) application of evolution to the DW. For solving (1) we use the transformation trace that was generated in the design. In order to solve (2) we propose an adaptation of the existing models and techniques for database schema evolution, to DW schema evolution, taking into account the features that differentiates the DWs from traditional operational databases.

Keywords

Data Warehouse (DW), DW evolution, Relational DW, DW design trace

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

1. Introduction

Source database schema may change, i.e. evolve. This invalidates the links between the source structures and the DW ones. Besides, the evolved database may have new data available that could be exploited by the DW. Therefore it is necessary to propagate source schema evolution to the DW.

The trivial solution for this problem would be re-designing all the DW. This implies starting from scratch, studying the problem and making design decisions again. However, the existence of the trace, which contains the design decisions, gives us the possibility of applying evolution to the DW.

In fact, the whole structure, composed by: trace, loading processes, DW schema and DW instance, has to evolve. However, we will see that evolution can involve changes over only one or some of these components.

In the cases where DW schema is changed, DW schema invariants have to be verified. In case these invariants are not satisfied, corrections to the schema have to be done (Consistency Corrections). After these corrections, the DW schema will be again in a consistent state. In addition, it will exist forward and backward conversion functions (f.c.f. and b.c.f.) (described in Chapter 2, Section 5) that are needed to transform data between old and new DW schema structures.

In **Figure 4.1** we show a global architecture of the evolution scenario in our context.

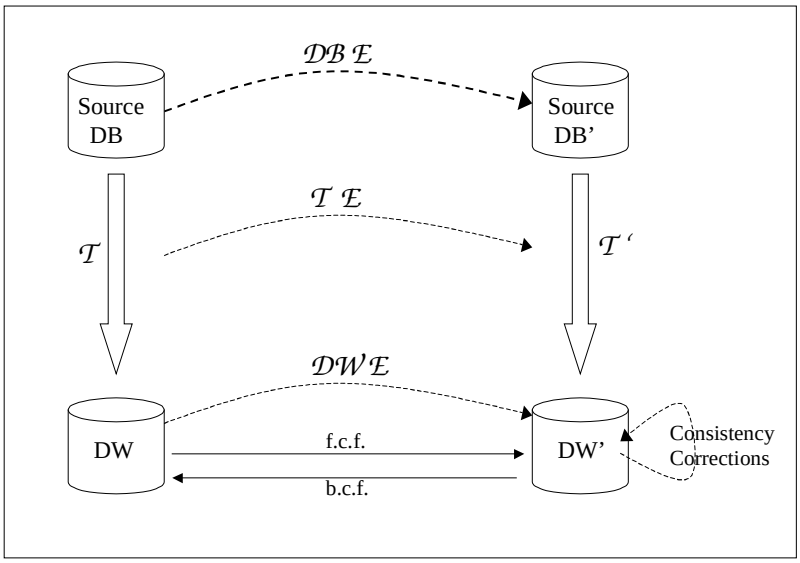


Figure 4.1

Data loading processes are generated from the trace, thus when we apply changes to the trace the associated data loading processes have to be re-generated.

The problem of propagating to the DW source schema evolution includes two main sub-problems: (1) **determining** the changes that must be done to the DW and to the trace, and (2) **applying** the corresponding changes to the DW and to the trace.

In Section 2 we present the Evolution Taxonomy of the source database, in Section 3 we present a solution for problem (1), in Section 4 we present a solution for problem (2), and in Section 5 we present the conclusion of this chapter. In **Figure 4.2** we show the structure of the chapter.

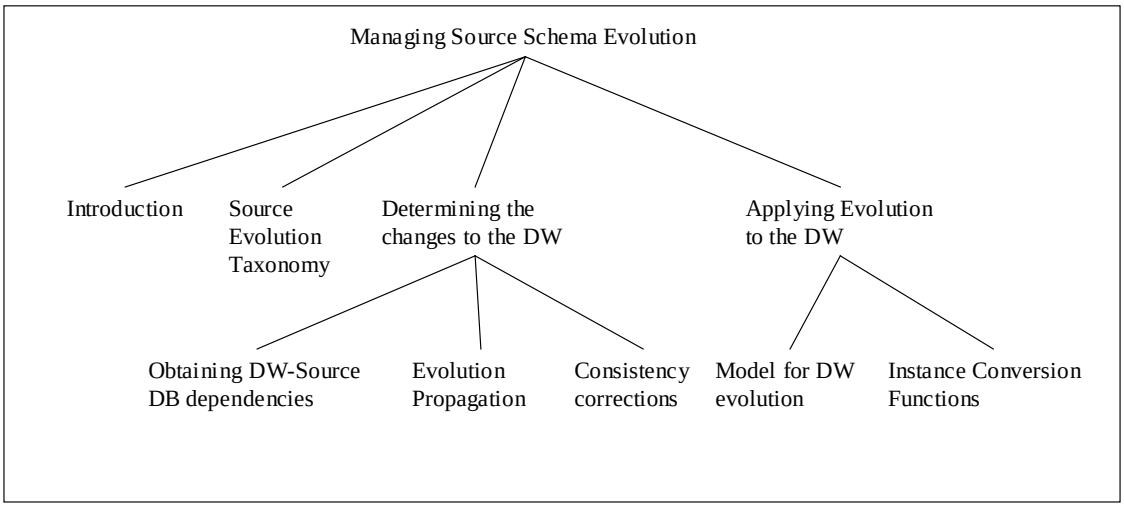


Figure 4.2

2. Source Evolution Taxonomy

In this section we define the taxonomy of changes that can happen to the source schema.

As we mentioned in Chapter 1, this work can be seen as a module of the project [CSI99] that is being developed in our research group. **Figure 4.3** shows the global architecture of the project. As can be seen, this work focuses on a part of the total process considered in the project. This part takes as input an integrated database. One of the other modules of the project [DoC00] solves the problem of propagation of source databases evolution to the integrated database.

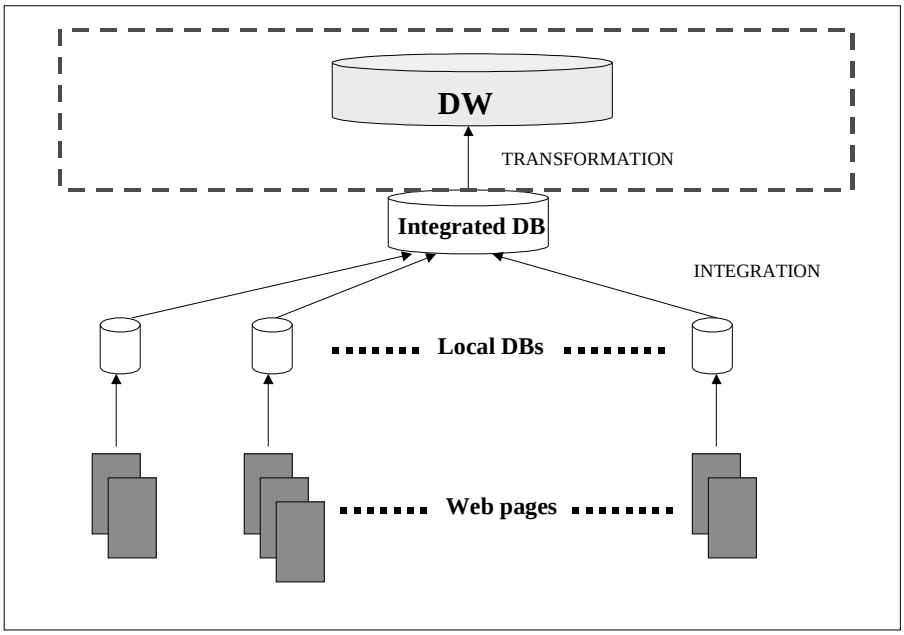


Figure 4.3

When there is evolution in one of the source databases, this is propagated to the integrated database, and then it must be propagated to the DW that was constructed from it. In the whole process considered in the project, the module that solves the problem of evolution of the integrated database would pass to our system the changes suffered by the integrated database and our system should propagate them to the DW. Therefore, our work should consider as the evolution taxonomy the set of schema changes that is managed by the mentioned module.

The taxonomy we use in this work covers the changes managed by the mentioned module of the project, presented in [DoC00]. However, it also includes some changes that are not considered in that module: *rename attribute*, *rename relation*, and *change the key of a relation*. These changes are added because they allow distinguishing more cases of change and provide more semantic to the evolution operations set. On the other hand, this taxonomy presents basically the same operations that are presented in taxonomies of the consulted bibliography [Zic91][Fer96][Ska86][Ban87].

The selected taxonomy for representing the possible changes to the source schema is the following:

1) Rename attribute
2) Add attribute
3) Remove attribute (the attribute cannot be a primary key)
4) Change the key of a relation
5) Rename relation
6) Add relation
7) Remove relation

3. Determining the changes to the DW

In this section we concentrate on the problem of determining the changes that must be applied to the DW and to the trace in order to propagate source schema evolution.

In this problem we have as input the trace and the change that has been applied to the source schema, and we have to give as solution the changes that must be applied to the DW and to the trace. The trace gives us the *dependencies* that exist between the source schema elements and the DW schema ones. We have to process the trace in order to deduce these dependencies.

The steps we follow for solving this problem are:

(a) definition of a mechanism for obtaining the dependencies between DW elements and source database elements
(b) analysis of the possible combinations of schema element dependencies and changes of the taxonomy
(c) definition of a set of *Propagation Rules* for each combination considered
(d) definition of a set of *Correction Rules* to be applied to the evolved schema for assuring its consistency

3.1. Obtaining DW-Source DB dependencies

The DW-Source DB dependencies we are most interested in are the ones between basic elements of the schemas, i.e. between attributes. Therefore, the first step we will perform in order to give a mechanism to deduce these dependencies is to express the primitives in terms of *basic operations* (operations that apply over basic elements of schemas).

Once we have de-composed the primitives into basic operations, we can process the trace by refining it, and obtaining the corresponding *detailed trace*. This is the trace in function of basic operations. After that, we can deduce the *dependency expressions* of a source schema element. A dependency expression gives the information of how an element of the DW schema depends on the selected element of the source schema. For example, an attribute of the DW schema could be a calculation from an attribute of the source schema.

In following sub-sections we present the set of basic operations, the primitives expressed in function of them, and the processing of the trace that is applied for obtaining the dependency expressions of the elements.

3.1.1. Basic operations

The transformation primitives can be de-composed into smaller operations that apply over basic elements of the sub-schemas. We define a set of *Basic Operations* that apply over basic elements of the data model we use, and that cover all the changes the primitives may do over these elements. Therefore, the primitives defined can be expressed in terms of these basic operations.

We classify the operations according to what object they are modifying.

During the schema transformation process, a set of relational elements (relations with all their properties) is maintained. This set is the intermediate result corresponding to each step of the process. We call the current intermediate result, *the context*.

The set of Basic Operations is shown in **Figure 4.4**.

Applied to	**Operations**	**Description**
The Context	Rel_add	Add a relation.
	Rel_del	Remove a relation.
A Relation	Att_add	Add a set of attributes.
	Att_rem	Remove a set of attributes.
	Att_cpy	Copy a set of attributes from a relation.
	Att_calc	Add a derived attribute.
A set of keys	Key_add	Add a key.
	Key_del	Remove a key.
A set of foreign keys	Fkey_add	Add a foreign key.
	Fkey_del	Remove a foreign key.

Figure 4.4

When we substitute a primitive by the sequence of basic operations, we lose the abstraction of the primitive. This abstraction is essential at the moment of design, but it is not important when considering the trace of the design made.

In Appendix 2 we provide the list of the basic operations with their descriptions.

Notation: *Basic_operation_Name* is the set of the names of the Basic Operations.

3.1.2. The Primitives expressed in terms of basic operations

We expressed the transformation primitives in terms of the basic operations that were previously defined. The set of primitives specified through these operations is presented in Appendix 3.

3.1.3. Processing the transformation trace

The design trace of the DW schema provides a mapping between original and final schema elements. It allows us to identify certain elements of the source schema and know the transformation they suffered during the DW schema design.

Using the trace we can identify certain element in the source schema and know all the operations that were applied to it during the schema transformation process, obtaining the *transformation trace of the element*. Then, starting from this trace we can obtain the *dependency expressions of the element* (defined later in this section), where elements of the DW schema are expressed in function of the source schema element.

In this section we concentrate in defining a mechanism to process the design trace, with the ultimate goal of obtaining the dependency expressions of the source elements.

Given an element of the source schema that has changed, we have to follow three steps with respect to the design trace:

1) Extract from the design trace the *transformation trace of the element* in terms of primitives.
   The transformation trace of the element in terms of primitives contains the set of the sequences of primitives that were applied to the element. We consider that a sequence of primitives was applied to an element if this element was part of the input of the first primitive of the sequence.

   It does not matter if the considered element is a relation or a part of one, the extracted trace will always be the trace of a relation, since the input schema of the primitives is always a set of relations.

2) Obtain the *detailed trace of the element*.
   Express the trace obtained in (1), in terms of basic operations. Extract an expression that shows only the sequence of basic operations that were applied to the considered element.

3) Obtain the *dependency expressions of the element*.
   From the detailed trace of the element we deduce its dependency expressions.

Following subsections present the used notation and mechanisms to obtain: the detailed trace and the dependency expressions of an element.

Detailed trace of an element

In order to obtain the detailed trace of an element from its trace, we have to do an "explosion" of the primitives that appear in the trace, de-composing them into the basic operations they perform.

With respect to the graphical representation of this trace, the idea is to explode each circular node (circular nodes represent primitives) into a set of nodes that represent the basic operations performed by the primitive. At the same time the rectangular nodes (corresponding to relations) must be exploded into sets of nodes that represent the sets of attributes of the relations. The obtained diagram is the graphic representation of the detailed trace of the element.

We can apply the same idea to the textual representation of the trace. The textual representation of the trace in terms of primitives consists of functional expressions. When we explode the primitives into the corresponding basic operations, we do not preserve this "functional format" of the expressions. We express the detailed trace of each relation as a sequence of basic operations applications.

Definition: Detailed Trace of a relation $T_D(R)$

Given a set of relations, a set of attributes, a set of functions and a set of basic operations, the Detailed Trace of a relation is represented by the following grammar:

$T_D(R)$ ::= <opapp_seq>
<opapp_seq> ::= <op_app> | <op_app>
<opapp_seq>
<op_app> ::= <operation> "(" <arg_list> ")"
<operation> ::= *Basic_operation_Name*
<arg_list> ::= <argument> | <argument> ',' <arg_list>
<argument> ::= <relation> | <att_set> | <att_set_set> | <function>
<relation> ::= *Rel_Name*
<att_set_set> ::= '{' <att_sets> '}'
<att_sets> ::= <att_set> | <att_set> ',' <att_sets>
<att_set> ::= '{' <attributes> '}'
<attributes> ::= <attribute> | <attribute> ',' <attributes>
<attribute> ::= *Att_Name*
<function> ::= *Fun_Name*

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

The <op_app> expression must respect the format of the input of the basic operation, which is stated in the specification of the basic operation.

♦

The textual representation of the detailed trace of a relation is the representation that best allows us to deduce the detailed trace of an attribute of the relation. Exploring this trace we can extract exactly the sequence of basic operations that were applied to the attribute.

For the representation of the detailed trace of an attribute we define a graph $G(T_{att})$.

Definition: Detailed Trace of an attribute. Graph $\boldsymbol{G(T_{att})}$.

Given a set of relations, a set of attributes, a set of functions and a set of basic operations, the Detailed Trace of an attribute is represented by the graph $G(T_{att})$, with the following characteristics:

The nodes represent attributes or the null value. The edges represent the application of a basic operation that transforms one attribute into the other. The edges have labels that are the names of the corresponding operation. It exists a path between two attributes when it is possible to reach one from the other going through the edges.

$G(T_{att}) = < Nodes, Edges, Paths >$

- $\forall$ n $\in$ *Nodes*, *Att*(n) returns the attribute represented by the node.

- Let $n_1, n_2 / n_1, n_2 \in$ *Nodes*, $\exists$ e(n_1, n_2) $\in$ *Edges* $\Leftrightarrow$ *Att*(n_2) = bop(*Att*(n_1)),

bop $\in$ *Basic_Operations*,

- Let $n_1, n_2 / n_1, n_2 \in$ *Nodes*, $\exists$ p(n_1, n_2) $\in$ *Paths* $\Leftrightarrow$

$\exists$ e(n_1, m_1), e(m_1, m_2), e(m_2, m_3), …, e(m_N, n_2) $\in$ *Edges*

♦

The general format of the graph is as follows:

<attribute1> —<op1>→ <attribute2> —<op2>→ ..... —<op3>→ <attributen>

We illustrate the proposed mechanisms through an example.

Example:

Consider the example trace presented in Chapter 3, Section 7.1. Suppose we are interested in the trace of the attribute **quantity** of the relation **SALES**. The detailed trace of the relation SALES is obtained from its transformation trace, decomposing the primitives that are part of this trace into the basic operations they perform.

<u>The trace of SALES:</u>

Graphical and textual representations are shown in **Figure 4.5** and **Figure 4.6**.

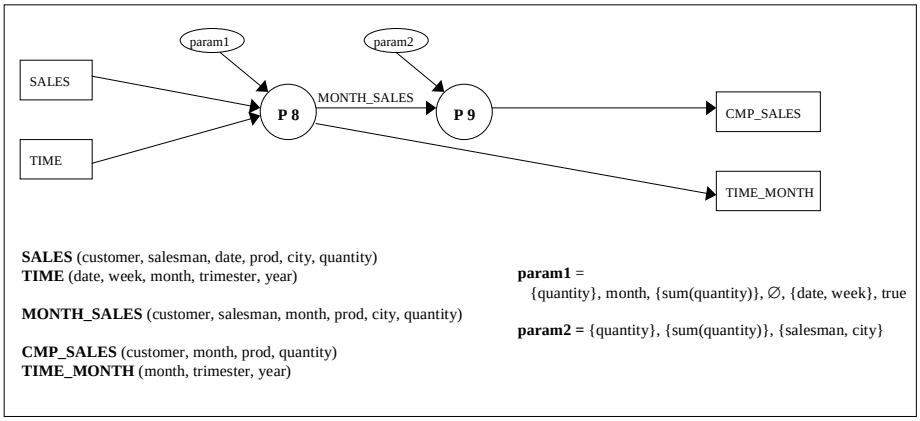


Figure 4.5

{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})

Relation schemas:

SALES (customer, salesman, date, prod, city, quantity)
TIME (date, week, month, trimester, year)
MONTH_SALES (customer, salesman, month, prod, city, quantity_m)
CMP_SALES (customer, month, prod, quantity_cmp)
TIME_MONTH (month, trimester, year)

Figure 4.6

The detailed trace of relation SALES:

Graphical and textual representations are shown in **Figure 4.7** and **Figure 4.8**.

As can be seen, graphical representation for detailed traces does not seem to be so practical; it becomes difficult to manage because of the large amount of elements it has to represent. This representation may be more manageable if it is restricted to a small portion of the whole trace.

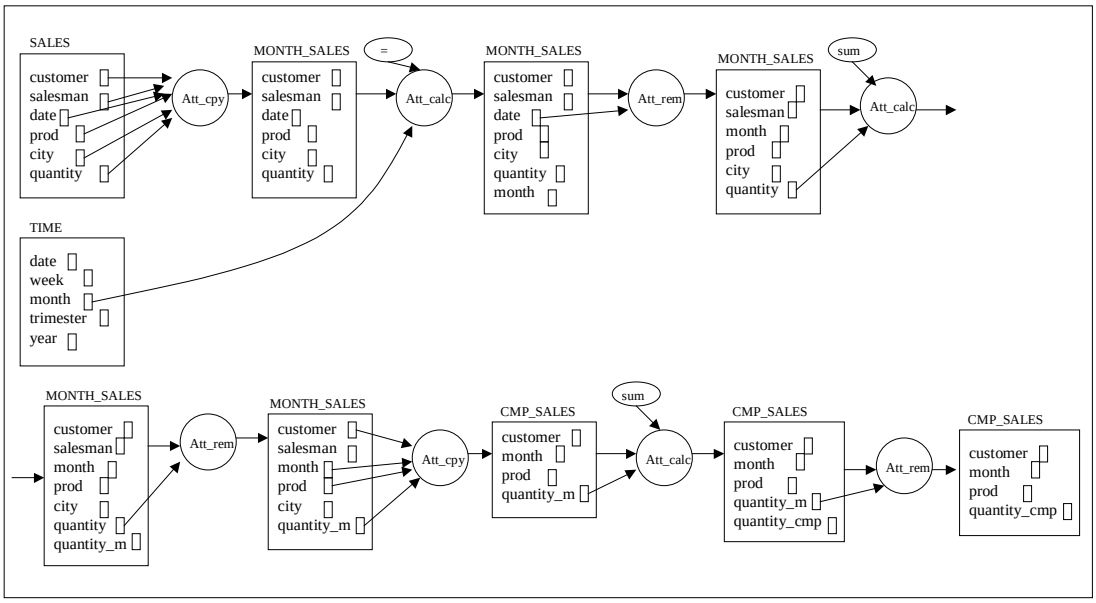


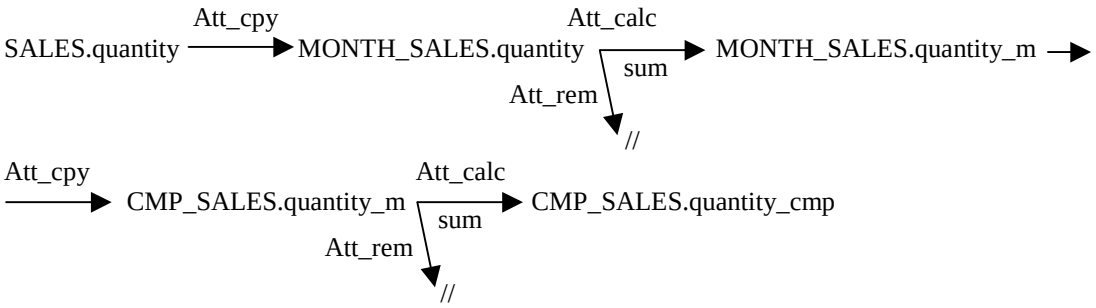
Figure 4.7

MONTH_SALES trace = **Att_cpy** ({customer, salesman, date, prod, city, quantity}, SALES, MONTH_SALES)
Att_calc ({TIME.month}, =, {TIME.date, MONTH_SALES.date}, MONTH_SALES.month)
Att_rem ({date}, MONTH_SALES)
Att_calc ({MONTH_SALES.quantity}, sum, {}, MONTH_SALES.quantity_m)
Att_rem ({quantity}, MONTH_SALES)

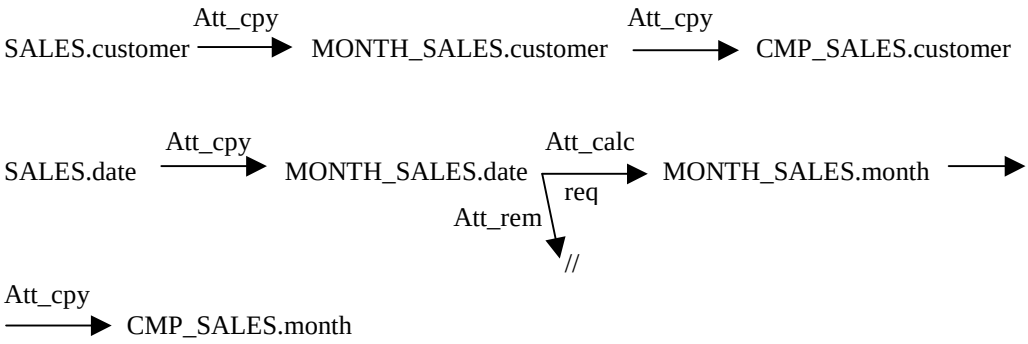
CMP_SALES trace = **Att_cpy** ({customer, month, prod, quantity_m}, MONTH_SALES, CMP_SALES)
Att_calc (CMP_SALES.quantity_m, sum, {}, CMP_SALES.quantity_cmp)
Att_rem ({quantity_m, CMP_SALES)

Figure 4.8

From the textual representation of the detailed trace of SALES we can easily extract the detailed trace of the attribute SALES.quantity:



Other examples are the traces of the attributes customer and date:



Note: In this representation, when the operation is Att_calc we also specify the calculation function that is used. We use the word "req" when the attribute is required for the calculation although it does not participate directly in the function.

♦

Dependency expressions of an element

The last step we have to follow in the processing of the trace of an element is to obtain the dependency expression of the element. This is an expression of the final element in function of the original one.

The dependencies information required for the management of source schema evolution vary according to the type of element considered (attribute or relation). Therefore, the dependency expressions that are constructed for each type of element will have different formats.

If the element is an attribute, the possible operations that can have been applied to it are: a copy, a calculation, and a remove. The dependency expression of an attribute will be deduced from its trace, considering the combination of copies and calculations. The removes do not participate in the generation of the dependency expressions.

If the element is a relation, the information needed about its dependencies is related to the dependencies of its attributes. Thus, a dependency expression of a relation with respect to a final relation, should specify the number of attributes that are copied to the final relation, and the number of attributes that participate in derivations of attributes of the final relation.

First we will present the dependency expressions for attributes and then the dependency expressions for relations.

Dependency expression of an attribute:

Simple dependency expressions:

Trace	Dep. expression
$R_1.A_1 \xrightarrow{\text{Att\_cpy}} R_2.A_2$	$R_2.A_2 = R_1.A_1$
$R_1.A_1 \xrightarrow[F]{\text{Att\_calc}} R_2.A_2$	$R_2.A_2 = f(R_1.A_1)$

In most cases the trace of an attribute will consist of a sequence of operation applications, causing the generation of a complex dependency expression. In these cases the dependency expression for the attribute must be constructed composing the operation applications.

Mechanism to construct a complex dependency expression:

<left_part> = <right_part>

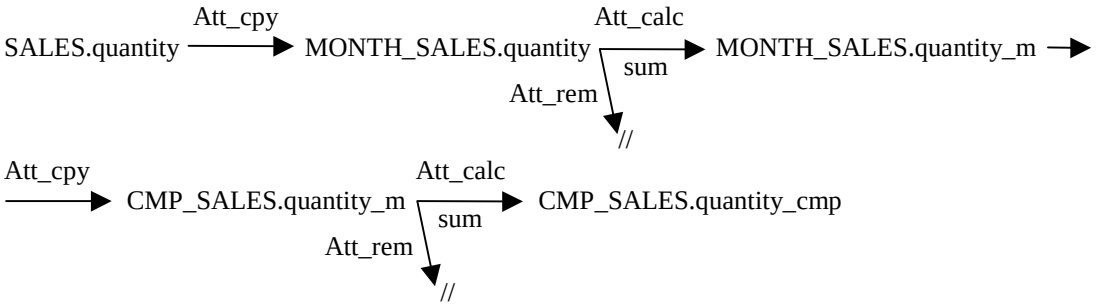
1) <left_part>: Left part of the expression: Last element of the trace. This element belongs to the final schema.

2) <right_part>: Right part of the expression: Follow the trace starting from the final element. Substitute each attribute of the trace by the corresponding expression according to the simple dependency expressions presented below, until an expression in function of the first attribute of the trace is obtained.

Note that in the case of calculation dependencies this expression shows how a final element depends on a source element, but it does not mean that the final element depends exclusively on this source element; it may depend also on other attributes.

Examples:

Trace of attribute SALES.quantity:



Dependency expression of attribute SALES.quantity:

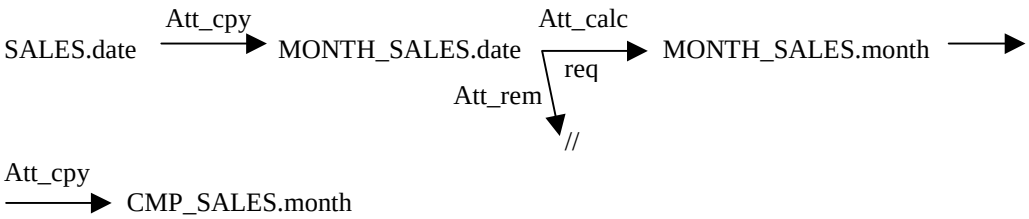
CMP_SALES.quantity_cmp = sum (sum (SALES.quantity))

Trace of attribute SALES.customer:

Dependency expressions of attribute SALES.customer:

CMP_SALES.customer = SALES.customer

Trace of attribute SALES.date:



Dependency expressions of attribute SALES.date:

CMP_SALES.month = req (SALES.date)

Note: Looking at the detailed trace of SALES we can see that the attribute CMP_SALES.month also depends on other attributes: TIME.date and TIME.month.

♦

Dependency expression of a relation:

Mechanism to construct a dependency expression between a source and a final relation:

1) Make a list containing all the dependency expressions of all the attributes of the source relation with respect to the final relation.

2) Deduce from this list the number of attributes that are copied to the final relation and the number of attributes that are needed for the calculation of an attribute of the final relation.

3) Construct the dependency expression of the relation with the following format:

   <final_rel> = **dep_cpy** (<source_rel>, n) ∧ **dep_calc** (<source_rel>, m)

   where dep_cpy is an expression that indicates that n attributes are copied from <source_rel>, and dep_calc is an expression that indicates that m attributes of <source_rel> are used for the derivation of attributes of <final_rel>.

   Obtain a reduced dependency expression with the following format:

   <final_rel> = **dep** (<source_rel>, n+m)

Example:

Dependency expressions of the attributes of SALES:

CMP_SALES.customer = SALES.customer
CMP_SALES.month = req (SALES.date)
CMP_SALES.prod = SALES.prod
CMP_SALES.quantity_cmp = sum (sum (SALES.quantity))

Dependency expression of the relation SALES with respect to the relation CMP_SALES:

CMP_SALES = dep_cpy (SALES, 2) ∧ dep_calc (SALES, 2)

Reduced expression:

CMP_SALES = dep (SALES, 4)

♦

3.2. Evolution Propagation

Now that we have proposed a solution to the problem of determining the dependencies between final and initial schema elements (DW and source schema elements), we can focus on the problem of how changes on the source may be propagated to the DW schema.

In this Section our goal is to provide a set of *Propagation Rules* that state the modifications that should be applied to the DW schema after source schema evolution.

3.2.1. Deducing the Propagation Rules

Our goal in this section is to provide a set of *Propagation Rules* that give the modifications that have to be done to the trace and, when necessary, to the DW schema, when a change has occurred to the source schema. These modifications are stated according to: (i) the changes occurred to the source schema, and (ii) the dependencies between elements of the source schema and elements of the DW schema.

We will start by analysing the possible combinations change-dependency, determining in each case if the DW should be affected by the change or not. Each time the DW is affected by a change the trace will also be affected. However, sometimes the trace will be able to make the change to the source schema transparent to the DW. In these cases we will say that the trace "absorbs" the changes.

Afterwards, we will present the rules that will specify the actions to be performed for each combination change-dependency.

Analysing the "combinations change-dependency"

In the table in **Figure 4.9** we show the possible combinations between changes of the source schema and type of dependency of the involved source element with respect to the DW schema, pointing out whether the trace and/or the DW should be modified or not. At this stage, only the changes at attribute level are considered.

Dependency / **Change to source att.**	No dependency	Copied	Used in Calc.	Req. for Calc.
Rename attribute		T	T	T
Add attribute	T DW } ?			
Remove attribute		T DW	T DW ?	T DW
Change key of a relation		T ? DW		T DW ?

Figure 4.9

Note: In **Figure 4.9**, T represents the trace, a "?" symbol means that only in some cases the DW/trace must be modified.

If an attribute is **renamed** in the source schema, the trace should absorb this change. The attributes of the DW that depends on the renamed attribute of the source schema do not need to be changed in any case of dependency. Only the mapping between these DW attributes and the renamed attribute should be changed.

In the case of **adding** an attribute to the source schema, the repercussion to the DW schema cannot be decided automatically. The designer should participate in the decision and in the process of repercussion in case it exists. In order to allow this, the following questions should be made to the designer: (i) Do you want to add one (or more than one) corresponding attribute to the DW schema? (ii) Where and how do you want to add them? (iii) Do any of the new structures substitute any structure in the DW schema? Which one/s?

In case the answer of question (i) is "No", nothing has to be done to the DW nor to the trace, and questions (ii) and (iii) are not necessary. But if the answer is "Yes", then the designer has to answer questions (ii) and (iii). The mechanism we offer him for answering question (ii), is to apply transformations through application of primitives to the new attribute (and, if necessary, to other structures of the source schema), directly generating the new structures of the DW. Finally, answering question (iii), he has to specify if the new structures are substituting any structure of the DW and in this case which of them. If some structure is being substituted it is automatically eliminated. Obviously, if the answer of question (i) is "Yes" both the trace and the DW are modified.

When an attribute is **removed** from the source schema the three different cases of dependency have to be considered for deciding the repercussion this change will have. (a) If the attribute has a copy in the DW, this attribute of the DW has to be eliminated. This elimination can be implemented in different ways, for example not physically removing the attribute and stating a fixed null value for all its instances. Besides the trace has to be modified, removing the connections existing between the two eliminated attributes. (b) If the attribute is used in the calculation function of a derived attribute of the DW, then we propose two alternatives. One is to eliminate the derived attribute from the DW, and the other is to modify the calculation function of the derived attribute so that the removed source attribute does not participate any more in this function. In both cases the trace is modified and only in the first case the DW is modified. (c) If the attribute is required for the calculation of an attribute of the DW, the derived attribute must be eliminated. This is because an attribute is defined (in the trace) as required when it behaves as a "join attribute", i.e. it allows two relations to join in order to derive an attribute of one relation from attributes of the other relation. If this "join attribute" is lost, the calculation will no longer be able to be done. In this case both the trace and the DW must be modified.

Now we will consider the case of **changing the key** of a relation, combining it with some of the possible existing dependencies between source and DW attributes. (A) When the source attribute that is the "old" key has a dependency of copy in the DW, there are two possibilities for the corresponding DW attribute: (i) it is key in a DW relation, and (ii) it is foreign key in a DW relation. In case (i) the DW must be modified changing the key so that it agrees with the "new" key defined in the source schema. If the attribute defined as "new" key does not exist in the DW relation, then it must be added. Only in case of adding an attribute the trace must be modified. In case (ii), the attribute corresponding to the "old" key defined as foreign key in the DW relation, must be substituted by the attribute that corresponds to the "new" key in the source. In the trace we have to delete the path corresponding to the substituted DW attribute and add the path corresponding to the added DW attribute. Both DW and trace must be modified. (B) When the source attribute that corresponds to the "old" key is used in the calculation function of a DW derived attribute, no action has to be performed, since the change should not affect the DW or the trace. (C) When the source attribute that corresponds to the "old" key is required for the calculation of a DW derived attribute, user participation is needed for deciding the repercussion the change will have. As said below, an attribute is defined (in the trace) as required when it behaves as a "join attribute" with respect to other relation. Therefore, we give two alternatives to the user: (i) eliminate the derived attribute in the DW, and (ii) substitute in the trace the required attribute by the "new" key attribute, paying attention to also changing the corresponding join attribute of the other relation. In (i) both the trace and the DW are modified, while in (ii) only the trace is modified.

Dependencies between relations

When we consider the changes of the taxonomy that affect a whole relation instead of an attribute, we can take into account the dependency that exists between a source relation and the DW relations. The dependency expression between two relations tells "how much" the DW relation is derived from the source one. This information can be useful for deciding if it is worthwhile to maintain a DW relation when the corresponding source relation was removed.

The dependency between a DW relation and a source relation is reduced to how many attributes of the DW relation depend on the source relation. We define a parameter *t*, to be set by the user, that states a threshold for this quantity. This value will be used in the corresponding Propagation Rules.

Propagation Rules

These rules state the actions that must be performed in each case of change to the source Database and dependency between source and DW elements.

For specifying the actions that affect the DW we use the Basic Operations defined in Section 3.1.1, since these operations work over a database schema and at a level that is suitable for the actions that must be performed. In addition, the use of the Basic Operations facilitates the specification of the *Consistency Corrections* for satisfying the invariants, which will be presented in next section.

R1) **CHANGE:** Rename attribute: A1 -> A2, where A1, A2 ∈ *Att_Name*

DEPENDENCY: Copied, Used in calculation, or Required for calculation

ACTION: - substitute in $G(T_{att})$ A1 by A2.

R2) **CHANGE:** Add attribute

DEPENDENCY: None

ACTION: - if user wants to add sub-schema DW_{SS} to the DW schema
- user applies primitives adding DW_{SS}
- if user wants to remove an existing DW sub-schema, DW_{SS}'
- for each R ∈ DW_{SS}'
- Rel_del (R) // remove from DW relation R
- remove path(_,A) from $G(T_{att})$, where A ∈ R // Remove form trace all paths that finish on an R's attribute.

R3) **CHANGE:** Remove attribute R.A

DEPENDENCY: Copied. R'.B = R.A

ACTION: - Att_rem ({B}, R') // remove from DW schema attribute B
- remove path(A,B) from $G(T_{att})$ // remove from trace path(A,B)

R4) **CHANGE:** Remove attribute R.A

DEPENDENCY: Used in calculation function. R'.B = f(R.A)

ACTION: - if user wants to remove attribute R'.B
- Att_rem ({B}, R') // remove from DW schema attribute B
- remove path(A,B) from $G(T_{att})$ // remove from trace path(A,B)
- else
- remove path(A,B) from $G(T_{att})$ // remove from trace path(A,B)
- user modifies the calculation function of B in the trace.

R5) **CHANGE:** Remove attribute R.A

DEPENDENCY: Required for calculation. R'.B = req(R.A)

ACTION: - Att_rem ({B}, R') // remove from DW schema attribute B
- remove path(A,B) from $G(T_{att})$ // remove from trace path(A,B)

R6) CHANGE: Change the key of a relation R. old key = A, new key = A'.

DEPENDENCY: Copied. R'.B = R.A.

ACTION: - if B is key in DW relation R'

- if ∃ B'∈ R', R'.B' = R.A'
  - Key_del ({B}, $Att_K(R')$)
  - Key_add ({B'}, $Att_K(R')$) // set B' as the key of R' in the DW
- else
  - Att_add ({B'}, R') // add attribute B' to relation R' in the DW
  - Key_del ({B}, $Att_K(R')$)
  - Key_add ({B'}, $Att_K(R')$) // set B' as the key of R' in the DW
  - add path(A',B') to $G(T_{att})$ // add path(A',B') to the trace
- else if B is foreign key in DW relation R', with respect to DW relation R''
  - Att_add ({B'}, R') // add attribute B' to relation R' in the DW
  - FKey_add ({B'}, $Att_{FK}(R',R'')$, $Att_{FK}(R')$) // set B' as foreign key to R'' in the DW
  - Att_rem ({B}, R') // remove attribute B from R' in the DW
  - remove path(A,B) from $G(T_{att})$ // remove from trace path(A,B)
  - add path(A',B') to $G(T_{att})$ // add path(A',B') to the trace

R7) CHANGE: Change the key of a relation R. old key = A, new key = A'.

DEPENDENCY: Required for calculation. R'.B = req(R.A)

ACTION: - if user wants to eliminate attribute B from DW

- Att_rem ({B}, R') // remove attribute B from R' in the DW
- remove path(A,B) from $G(T_{att})$ // remove from trace path(A,B)
- else if user wants to change the required attribute in the trace
  - substitute path(A,B) by path(A',B) in $G(T_{att})$
  - user corrects path(_,B), updating the other required attributes.

With rules R1 to R7 we cover the changes of the Taxonomy that affect an attribute (the first four changes). Rules R8 to R10 cover the changes over a whole relation (the last three changes of the Taxonomy).

R8) CHANGE: Rename relation: R1 -> R2, where R1, R2 ∈ *Rel_Name*

DEPENDENCY: R = dep (R1, n), ∀ n

ACTION: - substitute in $G(T_{att})$ R1 by R2

R9) CHANGE: Add relation: R

DEPENDENCY: None

ACTION: - if user wants to add sub-schema DW_{SS} to the DW schema

- user applies primitives adding DW_{SS}
- if user wants to remove an existing DW sub-schema, DW_{SS}'
- for each $R \in DW_{SS}$'
- Rel_del (R) // remove from DW relation R
- remove path(_,A) from $G(T_{att})$, where $A \in R$ // Remove form trace all paths that finish on an R's attribute.

R10) CHANGE: Remove relation: R

DEPENDENCY: R' = dep (R, n), where $n > t$

ACTION: - Rel_del (R') // remove from DW relation R'

- remove path(_,A) from $G(T_{att})$, where $A \in R'$ // Remove form trace all paths that finish on an attribute of R'.

R11) CHANGE: Remove relation: R

DEPENDENCY: R' = dep (R, n), where $n <= t$

ACTION: - for each $A / A \in Att(R) \wedge A \in Att(R')$

- if R'.A = R.A
- apply **R3**
- else if R'.A = f(R.A)
- apply **R4**
- else if R'.A = req(R.A)
- apply **R5**

3.3. Consistency corrections

When a Database schema is modified it may happen that some property that was satisfied by the schema before the change, is not satisfied after the change. In Chapter 3, Section 3 we have defined a set of consistency properties that must be satisfied by a DW schema, which we called *invariants*.

In the previous section we proposed the Schema Propagation Rules for propagating source schema evolution to the DW schema. However, once changes to the source schema were propagated to the DW schema, an important task has to be carried out yet: the verification of DW schema consistency and, if necessary, its correction. **Figure 4.10** shows an example, which is explained following. In a) *Sales* is a source relation, *Sales_DW* is a DW relation (a measure relation), and $\mathcal{T}$ is the trace that relates them. In b) the schema of *Sales* changes. Attribute *city_id* is removed. In c) schema evolution is propagated to the DW. Attribute *city_id* is removed form relation *Sales_DW* and $\mathcal{T}$ is modified. However, relation *Sales_DW* still contains an attribute, *city_name*, that makes it inconsistent according to the "measure relations invariant". In d) *city_name* is removed and $\mathcal{T}$ is modified, so that *Sales_DW* satisfy the invariants.

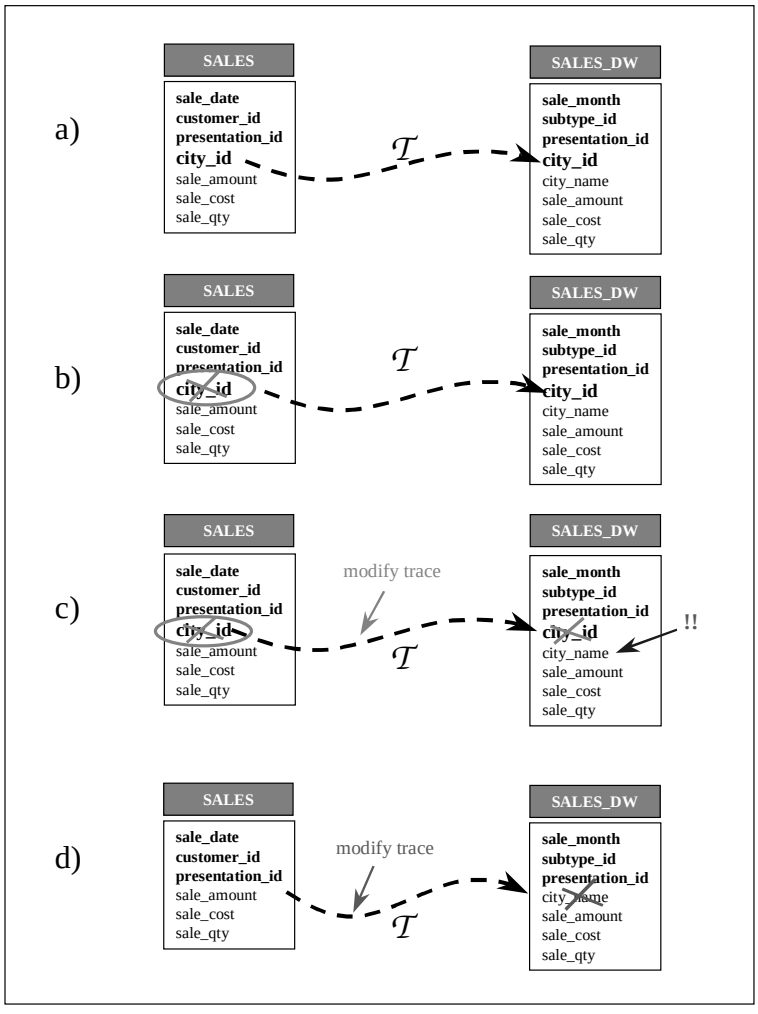


Figure 4.10

In this section we propose a mechanism to correct the DW schema in case the changes applied to it have left it in an inconsistent state, i.e. in case the DW schema does not satisfy the DW schema invariants any more. We provide a set of rules that intend to cover all the inconsistencies that may be generated by the DW evolution, and give the actions that should be performed in each case.

In this case we must consider the DW schema type of each element being changed. It will be relevant if, for example, a relation is of "measure" or of "dimension" type.

R1 – Foreign key updates

R1.1 - ON APPLICATION OF: *Key_del ({A}, $Att_K(R)$)* and *Key_add ({A'}, $Att_K(R)$)*, where A = old key and A'= new key

APPLY: *FKey_add ({A'}, $Att_{FK}(R_i,R)$, $Att_{FK}(R_i)$)* to all R_i / $Att_{FK}(R_i,R)$ = A

R1.2 - ON APPLICATION OF: *Rel_del (R)*, where $R \in Rel_D$

WHEN: $\exists$ R' $\in Rel_M$ / $Att_{FK}(R', R) \neq \emptyset$

APPLY: Primitive *Aggregate Generation* to R', removing X,

where X = { A / A $\in$ *Att(R)* $\wedge$ A $\in$ *Att(R')* }

R2 – Measure relations correction

ON APPLICATION OF: *Att_rem ({A}, R)* / $R \in Rel_M$

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

APPLY: *Att_rem ({B}, R)*

remove path(_,B) from $G(T_{att})$ // remove from trace the path that finishes in B

R3 – History relations update[1]

ON APPLICATION OF: *Att_add ({A}, R)*, obtaining A $\in$ *Att(R)*

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

APPLY: *Att_add ({A}, R')*, obtaining A $\in$ *Att(R')*

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

4. Applying evolution to the DW

In this section we focus on the problem of applying the corresponding changes to the DW and to the trace.

In order to solve this problem we have to: (a) define the model we will follow for the management of DW schema evolution, and (b) provide the *Conversion Functions* to be applied to the instance of the DW schema to transform it to an instance of the evolved DW schema.

4.1. Model for DW Evolution

In this section we define which strategy we would implement to apply evolution to the DW.

In Chapter 2, Section 5 we present an overview of the existing knowledge about schema evolution. In our proposal we extract some techniques from this existing work, and we adapt, combine and apply them for the resolution of our problem.

4.1.1. Previous considerations

We start enumerating the particular features of DWs, specially in the context of this work, that affect the treatment of evolution. Afterwards, we discuss how these elements affect the possible models or approaches considered in our work for applying evolution to the DW.

Some particular features of DWs:

- History data is stored in a DW.
- Applications that run over the DW only query the data. They do not modify it.
- Some evolution operations that in the context of operational databases are considered that do not corrupt existing applications using an adaptational approach [Fer96], in the context of DWs can lead to unexpected results.
- Most of the queries that are submitted to a DW require a big range of the history of the data existing in the DW.
- Due to the meta-information that our system manages, some of the conversion functions for the instances can be provided by it.

In a DW history data is relevant and it is maintained for a long time. Therefore, it would not be reasonable to transform this data to other formats perhaps loosing some of it or some of its semantic. Considering this aspect, a versioning approach would be a suitable solution.

We assume that modifications over the data only are applied in the context of loading data to the DW. For this reason, if we use the solution of schema versioning, only the last version will be updated. It will never exist updates over the data of other versions; this data will only be queried. This situation is favourable for the application of versioning approach, because it will not be necessary to convert updates to the new format of the data into updates to the old format of the data, which seems to be a nontrivial problem.

In [Fer96] some schema update operations are classified as schema extending, and they are stated as not affecting existing applications in the context of an adaptational approach. These operations include, for example, "Create an attribute". Considering the DW evolution taxonomy we define in Section 4.2, the corresponding operation (doing a mapping between OODBs and RDBs) would be "Add attribute". We can show that in the specific case of *adding a foreign key to a measure relation*, this operation can lead to unexpected results of queries that run on the old schema. We show an example in **Figure 4.11**. Taking into account this difference, the proposal of integrating the two approaches [Fer96] does not seem to be so applicable to DW schemas.

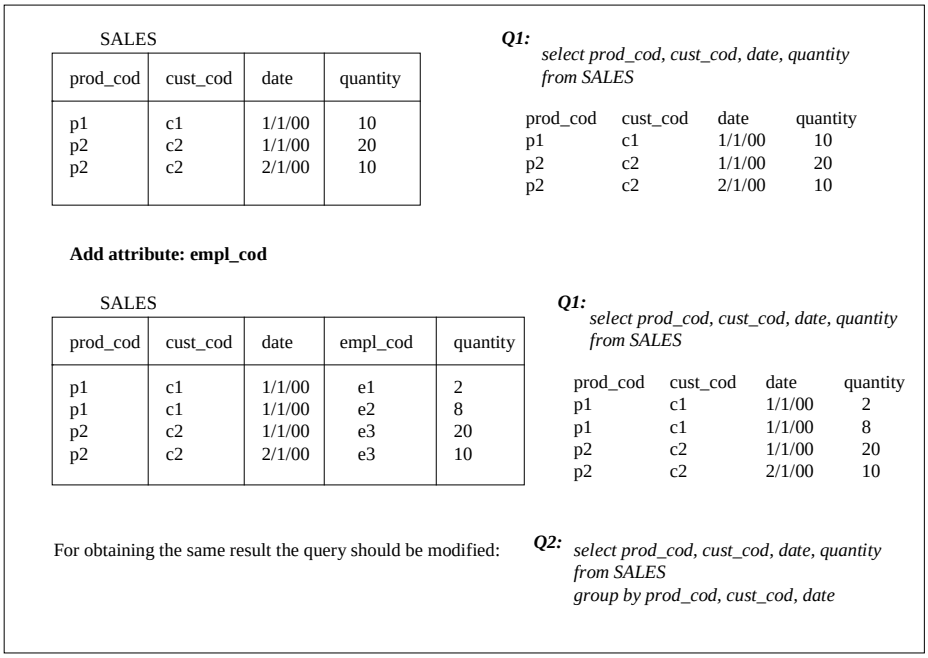


Figure 4.11

In general, queries that are submitted to a DW refer to data across a long time period. Therefore, if we work in a context of schema versioning, probably most of queries will require data of many different versions. In these cases the use of instance conversion functions will be necessary.

In this work we propose a context where a considerable amount of information about schemas and instances is maintained. This meta-information allows us to decide, in some cases, how data should be transformed in case of DW schema evolution. This is specified in Section 4.2 by the *instance conversion functions*.

4.1.2. The proposed mechanism

Considering the characteristics of the solutions extracted from the consulted bibliography, and the particular features studied in the previous section, we propose the following solution for applying evolution to a DW in our context:

Management of DW evolution is based on the versioning approach. We maintain a list of schema versions, as proposed in [Fer96]. We apply the same strategy for trace evolution.

The queries over the DW will behave according to the following guidelines:

- Queries that were already running over any version can continue running over the same version without any modification. These queries will not have access to information stored in subversions of that version.
- When a query is submitted to the actual (last) version, data stored in superversions is transformed through the f.c.f., which in some cases are provided by the system and in other ones are asked to the user. The mechanic is shown in **Figure 4.12**. The f.c.f are presented in Section 4.2 as i.c.f (instance conversion functions).

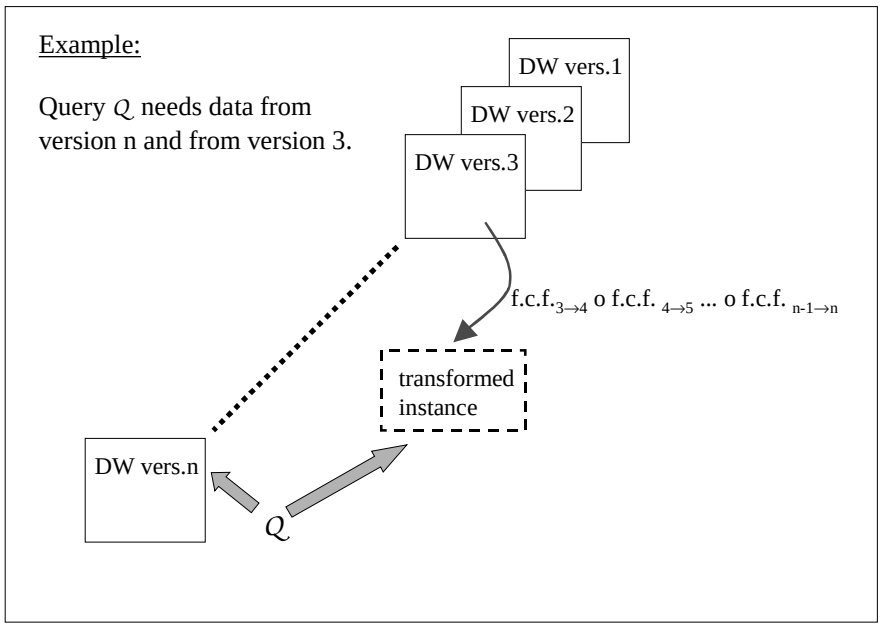


Figure 4.12

Note: If there are some queries to a version that need to access data of a newer version, it will be necessary to implement the b.c.f. for transforming this data.

4.2. Instance Conversion Functions

When an evolution operation has been applied to the DW schema, a conversion function can be applied to the instance of the old DW schema so that it can be seen as an instance of the evolved DW schema (see **Figure 4.13**).

In this section we provide the queries that have to be done to the data existing in the old DW in order to obtain the same data structured according to the new DW schema. We call these queries instance conversion functions (i.c.f.).

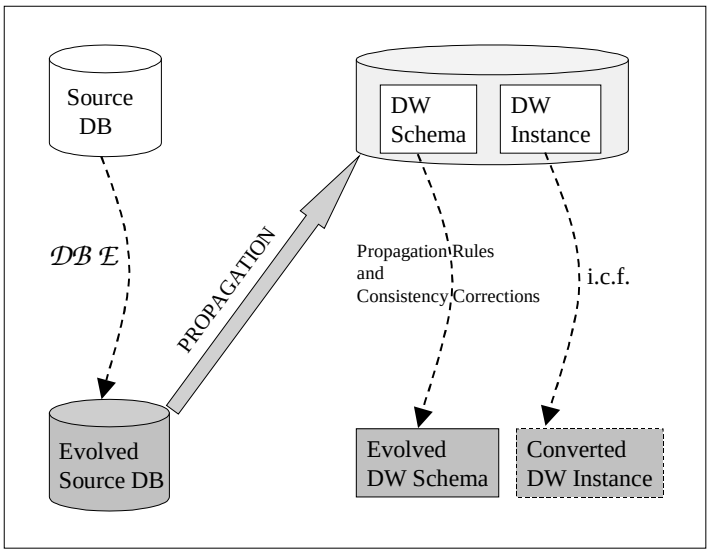


Figure 4.13

In some cases of change it is not possible to determine the i.c.f. automatically. For these cases we need the designer participation. Sometimes it is enough to ask the designer some questions, but other times is the designer who has to give the complete conversion function. The latter case happens when the change involves adding of information.

For determining the i.c.f. corresponding to each case of change, we must consider the type of the DW schema element that is being affected. In some cases, for example, the transformation of a relation instance will be different if the relation is a dimension or a measure one.

We define another taxonomy: a DW evolution taxonomy, which includes the possible changes that can be applied to the DW schema in our context. The changes are sub-classified according to the type of DW schema element, only in the cases that it is necessary to deduce the i.c.f.

DW Evolution Taxonomy

1) Add attribute
2) Remove attribute
   a) from Measure Relation
      a1) descriptive attribute
         - foreign key
         - not foreign key
      a2) measure attribute
   b) from Dimension Relation
      b1) descriptive attribute
      b2) hierarchical attribute
3) Change key of a relation
4) Change foreign key of a relation

Instance Conversion Functions

1) Add attribute

i.c.f. 1: user-defined function

2) Remove attribute
   c) from Measure Relation
      a1) descriptive attribute
         - foreign key

           i.c.f. 2: - $R \in Rel_M$, $A = Att_{FK}(R,R')$, $B_1, ..., B_n \in Att_M(R)$

           - provided by the user: list of $f_1(B_1), ..., f_n(B_n)$, where $f_1, ..., f_n$ are aggregation functions

           - select $\{Att_D(R) - A\}, f_1(B_1), ..., f_n(B_n)$
             from R
             group by $\{ Att_D(R) - A\}$

         - not foreign key

           i.c.f. 3: - $R \in Rel_M$, $A \in Att_D(R)$

           - select Att(R) - A
             from R

a2) measure attribute

i.c.f. 4: - R ∈ Rel_M, A ∈ $Att_M(R)$

- select Att(R) - A
  from R

d) from Dimension Relation

b1) descriptive attribute

i.c.f. 5: - R ∈ Rel_D, A = $Att_D(R)$

- select Att(R) - A
  from R

b2) hierarchical attribute

i.c.f. 6: - R ∈ Rel_D, A = $Att_J(R)$

- select Att(R) - A
  from R

3) Change key of a relation

i.c.f. 7: - R ∈ *Rel*, {A} ∈ $Att_K(R)$ old key, B ∈ *Att(R)* new key

- The instance must not be transformed

Note: It is not possible to define a conversion at this step. However, at the moment of query, the difference with respect to the keys should be considered.

4) Change foreign key of a relation

i.c.f. 8: - R ∈ *Rel*, {A} ∈ $Att_{FK}(R,R')$ old foreign key, B ∈ *Att(R)* new foreign key

- The instance must not be transformed

Note: It is not possible to define a conversion at this step. However, at the moment of query, the difference with respect to the keys should be considered.

5. Conclusion

This chapter focuses on the whole process that starts with evolution of the source schema and finishes with evolution of the DW schema.

We present a strategy that solves how to propagate the changes occurred on the source schema to the DW schema, and how to manage evolution in the context of the DW. The steps that should be performed in case of a change in the source schema are the following: 1- Identify the dependencies that exist between the changed element and elements in the DW. This is done using the trace (in Section 3.1). 2- Apply the Propagation Rules. Choose the appropriate rule according to the change and the dependency (in Section 3.2). Create a new schema if it has to be changed and a new trace. Mark them as a new version. 3- Verify the DW schema consistency and apply consistency corrections to the new schema if it is necessary (in Section 3.3). 4- Implement the f.c.f. for the instance, if it is possible (in Section 4.2). 5- If there is a new version of the schema or the trace, re-generate the loading processes. 6- Manage the queries as it is proposed in Section 4.1.2.

With respect to the classification of schema elements into DW elements, in the propagation rules it was not necessary to consider this classification, while in the instance conversion functions it had to be considered.

In Section 3.1.3 we present the detailed trace of an element and we define the graph of an attribute's detailed trace. We do not specify the procedure to pass from the detailed trace to this graph. We describe it, and we illustrate it with examples.

The Propagation Rules we propose state the modifications that must be done to the DW and to the trace. Another approach for this rules that seems to be more efficient for implementation is the following: The rules state only the modifications that must be done to the trace. At the moment of applying evolution the affected portion of the trace is re-applied (the operations of this portion of the trace are applied), generating the modified portion of the DW schema, which must substitute the original portion.

Bibliography

[Abe98] R. Abella, L. Coppola, D. Olave,. *Un Datawarehouse para la Facultad de Ingenieria.* Universidad de la República del Uruguay. In.Co. Proyecto de Taller 5. 1998.

[Ada98] C. Adamson, M. Venerable. *Data Warehouse Design Solutions.* J. Wiley & Sons, Inc. 1998

[Agr97] R. Agrawal, A. Gupta, S. Sarawagi. *Modeling Multidimensional Databases.* ICDE 1997

[Alc00] A. Alcarraz, M. Ayala, P. Gatto. *Diseño e Implementacion de una herramienta para evolucion de Data Warehouses.* Universidad de la República del Uruguay. In.Co. Proyecto en curso de Taller 5. 2000.

[Arz99] G. Arzua, G. Gil, S. Sharoian. *Manejador de Repositorio para Ambiente CASE.* Facultad de Ingenieria. Universidad de la República del Uruguay. In.Co. Proyecto de Taller 5. 1999.

[Bal98] C. Ballard. *Data Modeling Techniques for Data Warehousing.* SG24-2238-00. IBM Red Book. ISBN number 0738402451. 1998.

[Ban87] J. Banerjee, W. Kim, H-J. Kim, H. F. Korth. *Semantics and Implementation of Schema Evolution in Object-Oriented Databases.* In proc. of the ACM SIGMOD Int'l Conf. Management of Data, San Francisco, CA, May 1987.

[Bat92] Batini, Ceri, Navathe. *Conceptual Database Design. An Entity-Relationship Approach.* The Benjamin/Cummings Publishing Company, Inc. 1992

[Bla99-1] M. Blaschka. *FIESTA: A Framework for Schema Evolution in Multidimensional Information Systems.* Proc. of 6th. CAISE Doctoral Consortium, 1999, Heidelberg, Germany.

[Bla99-2] M. Blaschka, C. Sapia, G. Hofling. *On Schema Evolution in Multidimensional Databases.* Proc. DaWaK '99, Florence, Italy.

[Cal99] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, R. Rosati. (DWQ project). *A Principled Approach to Data Integration and Reconciliation in Data Warehousing.* Proc. CAISE '99 Workshop on Design and Management of Data Warehouses (DMDW '99), 1999.

[Cha97] S. Chaudhuri, U. Dayal. *An overview of Data Warehousing and OLAP Technology.* SIGMOD Record 26(1). 1997.

[CSI99] Grupo CSI. *Diseño y mantenimiento dinámico de Data Warehouses – Aplicación en el contexto de la Web*. V Jornadas de Informática e Investigación Operativa y VIII Encuentro del Laboratorio de Ciencias de la Computación . Facultad de Ingenieria. Universidad de la República del Uruguay. In.Co. Marzo '99.

[DoC00] A. do Carmo. *Aplicando Integración de Esquemas en un contexto DW-Web*. Master's Thesis. Pedeciba. Universidad de la República del Uruguay. 2000.

[Elm00] Elmasri, Navathe. *Fundamentals of Database Systems.* Addison-Wesley 2000.

[Fer93] F. Ferradina, R. Zicari. *Object Database Schema Evolution: are Lazy Updates always Equivalent to Immediate Updates?* Technical Report n11/93, University of Frankfurt. Presented at OOPSLA Workshop, September 1993, Washington D.C.

[Fer94] F. Ferradina, T. Meyer, R. Zicari. *Implementing Lazy Database Updates for an Object Database System.* Proc. of the 20th. International Conference on VLDB, Santiago de Chile, September 1994.

[Fer95] F. Ferradina, T. Meyer, R. Zicari. *Measuring the Performance of Immediate and Deferred Updates in Object Database Systems.* OOPSLA Workshop on Object Database Behaviour, Benchmarks and Performance. Austin, Texas, October 15, 1995.

[Fer96] F. Ferradina, S. Lautemann. *An Integrated Approach to Schema Evolution for Object Databases.* OOIS 1996, London, U.K.

[Gar99] P. Garbusi, F. Piedrabuena, G. Vazquez. *Diseño e implementación de una herramienta de ayuda en el diseño de un Data Warehouse Relacional.* Facultad de Ingenieria. Universidad de la República del Uruguay. In.Co. Proyecto de Taller 5. 1999.

[Gol98] M. Golfarelli, Stefano Rizzi. *A Methodological Framework for Data Warehouse Design.* DOLAP 1998.

[Hac97] M. S. Hacid, U. Sattler (DWQ project). *An Object-Centered Multi-dimensional Data Model with Hierarchically Structured Dimensions.* Proc. of the IEEE Knowledge and Data Engineering Workshop. 1997.

[Hai91] J. L. Hainaut. *Entity-Generating schema transformations for Entity-Relationship models.* ER 1991: 643 – 670.

[Ham95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, Yue Zhuge. *The Stanford Data Warehousing Project.* Data Eng. Bulletin, 18(2), June 1995.

[Hull97] R. Hull. *Managing Semantic Heterogeneity in Databases: A Theoretical Perspective.* PODS 1997.

[Hull96] R. Hull, G. Zhou. *A Framework for Supporting Data Integration Using the Materialised and Virtual Approaches.* SIGMOD Conf., Montreal, 1996.

[Inm96] W. H. Inmon. *Building the Operational Data Store.* John Wiley & Sons Inc., 1996.

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

[Kim96-2] R. Kimball. *Dangerous Preconceptions.* The Data Warehouse Architect, DBMS Magazine, August 1996, URL: http://www.dbmsmag.com

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

[Kim98] R. Kimball. *The Data Warehouse Lifecycle Toolkit.* J. Wiley & Sons, Inc. 1998

[Kor99] M. A. R. Kortnik, D. L. Moody. *From Entities to Stars, Snowflakes, Clusters, Constellations and Galaxies: A Methodology for Data Warehouse Design.* 18th. International Conference on Conceptual Modelling. Industrial Track Proceedings. ER'99.

[Lab97] W. J. Labio, Y. Zhuge, J. N. Wiener, H. Gupta, H. Garcia-Molina, J. Widom. Stanford University. *The WHIPS Prototype for Data Warehouse Creation and Maintenance.* SIGMOD 1997.

[Lab96] W. Labio, H. Garcia-Molina. *Efficient Snapshot Differential Algorithms for Data Warehousing.* VLDB Conf., Bombay, 1996.

[Lau96] S. Lautemann. *An Introduction to Schema Versioning in OODBMS.* In proc. of the 7th. Int'l. Conf. on Database and Expert Systems Applications (DEXA), Zurich, Switzerland, September 1996. IEEE Computer Society. Workshop Proceedings.

[Lau97] S. Lautemann. *Schema Versions in Object Oriented Database Systems.* In proc. of the 5th. Int'l. Conf. On Database Systems for Advanced Applications (DASFAA), Melbourne, Australia, April 1997.

[Lev96] A. Y. Levy, A. Rajaraman, J. J. Ordille. *Querying Heterogeneous Information Sources Using Source Descriptions.* VLDB 1996.

[Lig99] S. Ligouditianos, T. Sellis, D. Theodoratos, Y. Vassiliou. (DWQ project). *Heuristic Algorithms for Designing a Data Warehouse with SPJ Views.* Proc. DaWaK '99, Florence, Italy

[Ngu89] G. T. Nguyen, D. Rieu. *Schema Evolution in Object-Oriented Database Systems.* Data & Knowledge Engineering (DKE) , Volume 4, 1989.

[Nic98] A. Nica, A. J. Lee, E. A. Rundensteiner. *The CVS Algorithm for View Syncronization in Evolvable Large-Scale Information Systems.* In Proceedings of International Conference on Extending Database Technology (EDBT'98), Spain 1998.

[Pap96] Y. Papakonstantinou, S. Abiteboul, H. Garcia-Molina. *Object Fusion in Mediator Systems.* VLDB 1996.

[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 Técnico. 2000.

[Per99] V. Peralta, A. Marotta, R. Ruggia. *Designing Data Warehouses through schema transformation primitives*. 18th. International Conference on Conceptual Modelling. Posters and Demonstrations. ER'99.

[Pic99] A. Picerno, M. Fontan. *Un editor para CMDM.* Facultad de Ingenieria. Universidad de la República del Uruguay. In.Co. Proyecto de Taller 5. 1999.

[Qui99] C. Quix. *Repository Support for Data Warehouse Evolution.* Proc. CAISE '99 Workshop on Design and Management of Data Warehouses (DMDW '99), 1999.

[Run97] E. A. Rundensteiner, A. J. Lee, A. Nica. *On Preserving Views in Evolving Environments.* In Proceedings of 4th. Int. Workshop on Knowledge Representation Meets Databases (KRDB'97). Greece 1997.

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

[Ska86] A. H. Skarra, S. B. Zdonik. *The Management of Changing Types in an Object-Oriented Database.* OOP SLA 1986, Portland, Oregon.

[Sta90] B. Staudt Lerner, A. Nico Habermann. *Beyond Schema Evolution to Database Reorganization.* ECOOP/OOPSLA 1990 Proceedings.

[Theo99-1] D. Theodoratos, T. Sellis (DWQ project). *Designing Data Warehouses.* DKE '99

[Theo99-2] D. Theodoratos, S. Ligoudistianos, T. Sellis. (DWQ project). *Designing the Global Data Warehouse with SPJ Views.* Proc. CAISE '99, Heidelberg, Germany.

[Theo99-3] D. Theodoratos, T. Sellis. (DWQ project). *Dynamic Data Warehouse Design.* Proc. DaWaK '99, Florence, Italy

[Tho97] E. Thomsen. *OLAP Solutions. Building Multidimensional Information.* John Wiley & Sons, Inc., 1997.

[Tork97] M. Tork Roth, P. Schwarz. *Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources.* VLDB 1997.

[Vas99] P. Vassiliadis, M. Bouzeghoub, C. Quix. *Towards Quality-oriented Data Warehouse Usage and Evolution.* Proc. of the 11th. Conference on Advanced Information Systems Engineering (CAISE '99), Hiedelberg, Germany, 1999.

[Wid95] J. Widom. *Research Problems in Data Warehousing.* Int'l Conf. On Info. And Knowledge Management (CIKM), November 1995.

[Wie96] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, H. Garcia-Molina, J. Widom. *A System Prototype for Warehouse View Maintenance.* Workshop on Materialised Views: Techniques and Applications, June 1996.

[Wu97] Ming-Chuan Wu, Alejandro P. Buchmann. *Research Issues in Data Warehousing.* BTW German Database Conference, 1997.

[Zha99] Xin Zhang. *Data Warehouse Maintenance Under Interleaved Schema and Data Updates.* A master thesis submitted to the Faculty of the Worcester Polytechnic Institute. Thesis Advisor: Professor E. A. Rundensteiner. May 1999.

[Zhou95] G. Zhou, R. Hull, R. King, J. Franchitti. *Data Integration and Warehousing Using H2O*. Data Eng. Bulletin, 18(2), 1995.

[Zhu95] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom. *View Maintenance in a Warehousing Environment.* SIGMOD Conf., San Jose, May 1995.

[Zhu96-1] Y. Zhuge, H. Garcia-Molina, J. Wiener. *The Strobe Algorithms for Multi-source Warehouse Consistency*. PDIS, Miami Beach, 1996.

[Zhu96-2] Y. Zhuge, H. Garcia-Molina, J. Wiener. *Consistency Algorithms for Multi-Source Warehouse View Maintenance.* Technical report, Stanford University, 1996.

[Zic91] R. Zicari. *A Framework for Schema Updates In An Object-Oriented Database System.* GIP Altair, Politecnico di Milano, Milano, Italy, 1991.