PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

Diseño y Mantenimiento de Data Warehouse a través de Transformaciones de Esquema

Adriana Marotta

Octubre de 2000

Tutor: Raúl Ruggia

Tribunal: Alejandro Gutiérrez
Regina Motz
Nora Szasz

Diseño y mantenimiento de Data Warehouse a través de transformaciones de esquema.
Tesis de Maestría
Marotta, Adriana
I

ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 01-01
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, Octubre de 2000

Resumen

Un Data Warehouse (DW) es una base de datos que almacena información para satisfacer requerimientos de toma de decisiones. Es una base de datos que tiene características particulares en cuanto a los datos que contiene y en cuanto a su utilización.

En este trabajo nos concentramos en diseño de DW y evolución de DW.

Las características de los DWs hacen que el proceso y las estrategias de diseño de éstos sean diferentes de los usados para sistemas OLTP. Nosotros abordamos el problema de diseño de DW con un enfoque de transformación de esquemas. Proponemos un conjunto de primitivas de transformación de esquemas, las cuales son operaciones de alto nivel que transforman sub-esquemas relacionales en otros sub-esquemas relacionales. Junto con ellas proveemos algunas herramientas que pueden ayudar en el proceso de diseño de DW: (a) la traza del diseño, (b) un conjunto de invariantes de esquema de DW, (c) un conjunto de reglas que especifican como corregir situaciones de inconsistencia de esquemas que fueron generadas por aplicación de primitivas, y (d) algunas estrategias para diseñar el DW a través de aplicación de primitivas.

La evolución de esquema de un DW puede ser generada por dos causas diferentes: (i) un cambio en el esquema fuente o (ii) un cambio en los requerimientos del DW. En este trabajo abordamos el problema de evolución del esquema fuente. Separamos este problema en dos fases: (1) determinación de los cambios que deben ser aplicados al esquema de DW y a la traza, y (2) aplicación de la evolución al DW. Para resolver (1) utilizamos la traza de la transformación que fue generada en el diseño. Para resolver (2) proponemos una adaptación de los modelos y técnicas existentes para evolución de esquemas de bases de datos, a evolución de esquemas de DW, considerando las características que diferencian a los DWs de las bases de datos operacionales tradicionales.

Palabras clave

Data Warehouse, diseño DW, evolución esquema DW, transformación de esquemas, DW Relacional, traza diseño DW

Agradecimientos

Quisiera agradecer a mi tutor, el Profesor Raúl Ruggia, quien me guió durante todo el proceso de investigación y escritura de esta tesis. También quisiera agradecer a los Profesores Regina Motz, Alejandro Gutiérrez, y Nora Szasz, de quienes recibí valiosos aportes en distintas instancias de este trabajo, y a todos los integrantes del Grupo CSI por el apoyo que me brindaron.

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.

In this work we concentrate in DW design and DW evolution.

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.

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 design, DW schema evolution, schema transformation, Relational DW, DW design trace

Contents

CHAPTER 1. Introduction

1. Context

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.

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

2. Motivation

In this work we concentrate in DW design and DW evolution.

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.

Evolution in a DW can be generated by two different causes. A DW schema can evolve as a consequence of: (i) a change in the source schema or (ii) a change in the DW requirements. These two cases have to be treated separately, since they involve different taxonomies of changes and different processes to impact the DW schema.

Source schema evolution is particularly relevant in the cases where the DW is generated from Web sources. In this context source schema will probably change very frequently. Our research group is

[1] OLTP: On Line Transaction Processing

working on a project that covers the different stages that exist in a DW system which information is extracted from Web sites [CSI99]. The present work can be seen as a module of this project, although at the same time it is not specific to this context. In our work we have two main reasons to concentrate in the problem of source schema evolution: (1) the highly evolutive context of the project [CSI99], and (2) the important facilities for propagating schema changes from source to DW that are provided by our proposal for DW design.

3. Goal and proposed solutions

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.

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.

We separate the problem of propagating source schema evolution to the DW schema 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. This trace allows us to obtain the path that was followed by each schema element and then decide how to propagate the changes occurred on the source schema. In some cases it is not necessary to modify the DW schema, but we always have to modify the trace in order to maintain the connection between source and DW schema elements. We provide a set of propagation rules that state which changes have to be done to the DW and to the trace, depending on each case of source schema change and dependency between source and DW schema elements.

In order to solve (2) we analyse the applicability of existing schema evolution models and techniques to DW schema evolution. We consider DW features that affect the treatment of evolution. We adapt existing models, mainly applying the Versioning approach (presented in Chapter 2).

In addition, we propose instance conversion functions that are necessary to convert instances from one version of the DW to another. These functions are required for the posterior usage of the DW.

4. Contributions

This work contributes in two directions: (1) DW design and (2) DW evolution.

With respect to DW design, 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. There is an operational prototype, which covers the functionalities of DW design through primitive applications, and has been developed in the context of a graduate project [Gar99] and complemented in the context of this thesis (Chapter 5).

In DW evolution we also contribute mainly in two aspects. On one hand, we present a mechanism for deducing the changes that have to be done to the DW schema when the source schema evolves. This mechanism is designed for the context of DW design proposed in this work. On the other hand, we present an analysis of the applicability of database schema evolution techniques to DW schemas. There is an ongoing graduate project [Alc00], which will extend the existing prototype, including the functionalities of source schema evolution propagation.

5. Outline of the thesis

This thesis consists of six chapters. Chapter 2 presents an overview of the existing knowledge in the areas that are more relevant to our work. Chapter 3 and Chapter 4 contain our proposals: in Chapter 3 we present a solution for DW logical design and in Chapter 4 we present a solution for propagation of source schema evolution to the DW. Chapter 5 is a brief description of the implemented prototype for DW design. Chapter 6 presents the conclusions and future work. Finally, there are 4 appendices and the used bibliography.

CHAPTER 2. Existing knowledge

1. Introduction

Our work is related to various sub-areas of Databases research area. It is situated mainly in the area of DW, in particular DW Design and DW Evolution. However it also applies techniques of Schema Transformations and Schema Evolution. The base data model it uses is the Relational Model (for basic definitions about databases and Relational Model, see [Elm00]).

Existing DW design techniques were the base for the definition of the set of transformation primitives. In addition, existing knowledge about database schema evolution was almost directly applied to the definition of the model for DW schema evolution.

In this chapter we present an overview of the existing knowledge on the areas that are the most relevant to our work. In Section 2 we present an overview of DW problems and how they are addressed. In Section 3 we show the existing approaches and the existing practical techniques about DW Design. In Section 4 we enumerate some works about schema transformation. In Section 5 and Section 6 we present the existing knowledge about schema evolution and DW schema evolution. In Section 7 we present the conclusion of the chapter.

2. An overview of Data Warehousing

DW is a very wide research area. It has many different sub-areas and it can be treated with different approaches. Some overviews of the research area are [Wid95][Wu97][Cha97].

The global architecture of DW systems proposed or assumed in most works is the one shown in **Figure 2.1**, although there is a variant that is proposed in [Inm96]: the ODS (Operational Data Store), shown in **Figure 2.2**.

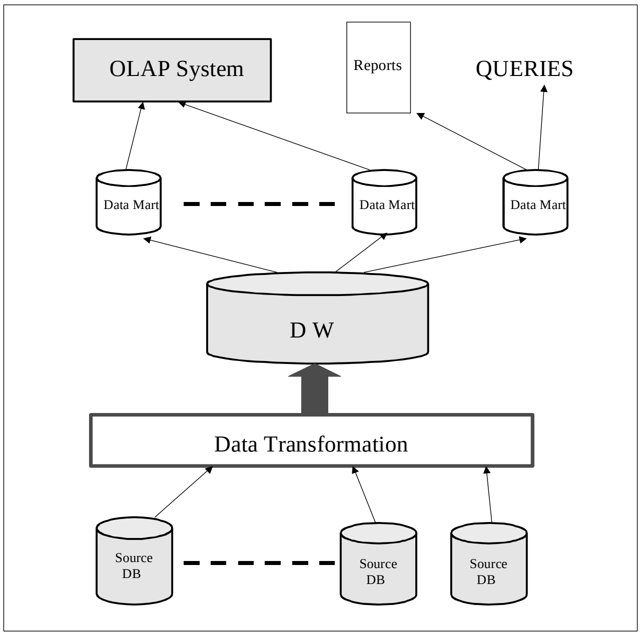


Figure 2.1

Source databases can be heterogeneous with respect to their data representation and to the data itself. Data integration is an important research area that attacks this problem. Some publications that concentrate on source heterogeneity are [Pap96][Lev96], which consider in particular the web sources. In many projects as H20, TSIMMIS, DWQ, strong attention is paid to data integration [Hull96][Hull97][Pap96][Cal99].

In order to translate heterogeneous data models to a common model, some authors propose the use of wrappers [Lab97][Tork97], which encapsulate data sources and mediate between them and the rest of the system.

Data transformation layer involves a wide range of transformations that have to be applied to source data, for example data quality control and data cleaning, data integration, and conversions that are necessary for adapting data to the DW structures.

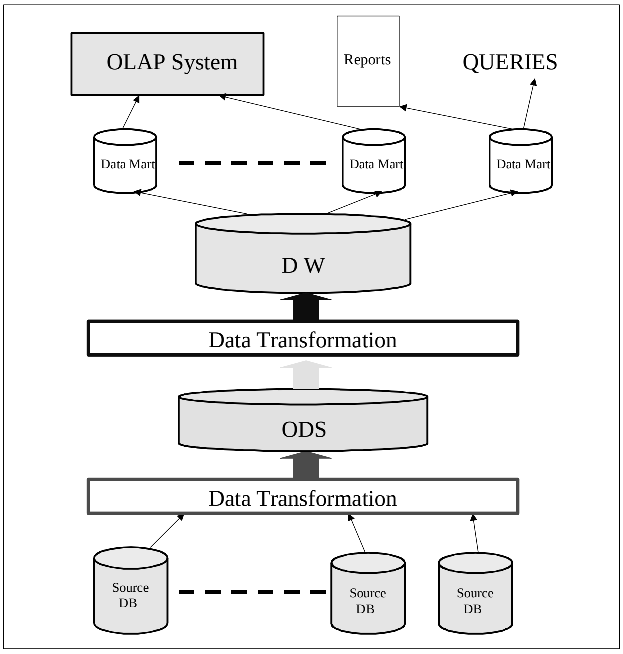


Figure 2.2

The ODS can be seen as an intermediate stage between the sources and the DW, although authors also propose that it can be used as a database for operational processing [Inm96][Kim96-2]. It contains integrated data, but this data is at detail level and it is only current data. Therefore we can think that with this architecture we are dividing the transformation work into two phases: in the first phase the main task is integration, and in the second phase the rest of the data transformation work is done.

Data Marts are proposed as logical subsets of the complete DW [Kim98]. They should be consistent in their data representation in order to assure DW robustness.

OLAP[2] [Tho97] systems have been heavily developed by industry community, while research community has not concentrated so much in it.

The data models that are used for DWs are Multidimensional Model and Relational Model. At the conceptual design level there is no discrepancy in choosing a multidimensional data model, since DW requirements are in general managed with a multidimensional perspective. Some publications about multidimensional data models are [Agr97][Gol98][Hac97]. The database system where the DW is built can be a multidimensional or a relational one. When this system is relational the logical design can be done applying techniques of multidimensional modelling to relational databases, as the ones presented in [Kim96-1]. This is the approach we adopted for data modelling in our work.

[2] OLAP: On Line Analytical Processing

The most used approach, in research community, for definition and management of the DW is the one of materialised views. In the WHIPS project [Lab97][Ham95][Wie96] they work mainly in definition and maintenance of the DW [Zhu95][Lab96] and view consistency [Zhu96-1][Zhu96-2]. In the H2O project [Zhou95] they propose the combination of materialised and virtual views and they focus on data integration [Hull96][Hull97]. A very recent proposal about materialised views is in [Theo99-1], where they address the problem of selecting the views to materialise. We comment more about this approach in next section.

3. DW Design

As we have shown in **Figure 2.1**, in some possible architectures, a DW may be used by an OLAP front-end or it may be queried directly by SQL statements.

We found in the literature, globally two different approaches for Relational DW design: one that applies *dimensional modelling* techniques, and another that bases mainly in the concept of *materialized views*.

Dimensional models represent data with a “cube” structure [Kim96-1], making more compatible logical data representation with OLAP data management. According to [Kor99], the objectives of dimensional modelling are: (i) to produce database structures that are easy for end-users to understand and write queries against, and (ii) to maximise the efficiency of queries. It achieves these objectives by minimising the number of tables and relationships between them. Normalized databases have some characteristics that are appropriate for OLTP systems, but not for DWs: (i) Its structure is not easy for end-users to understand and use. In OLTP systems this is not a problem because, usually end-users interact with the database through a layer of software. (ii) Data redundancy is minimised. This maximises efficiency of updates, but tends to penalise retrievals. Data redundancy is not a problem in DWs because data is not updated on-line.

The basic concepts of dimensional modelling are: *facts*, *dimensions* and *measures* [Bal98]. A *fact* is a collection of related data items, consisting of measures and context data. It typically represents business items or business transactions. A *dimension* is a collection of data that describe one business dimension. Dimensions determine the contextual background for the facts; they are the parameters over which we want to perform OLAP. A *measure* is a numeric attribute of a fact, representing the performance or behaviour of the business relative to the dimensions.

Considering Relational context, there are two basic models that are used in dimensional modelling: (i) *star model* and (ii) *snowflake model*. The *star model* is the basic structure for a dimensional model. It has one large central table (fact table) and a set of smaller tables (dimensions) arranged in a radial pattern around the central table. (We show an example in **Figure 2.3**). The *snowflake model* is the result of decomposing one or more of the dimensions. The many-to-one relationships among sets of attributes of a dimension can separate new dimension tables, forming a hierarchy. (**Figure 2.4** shows an example). The decomposed snowflake structure visualises the hierarchical structure of dimensions very well.

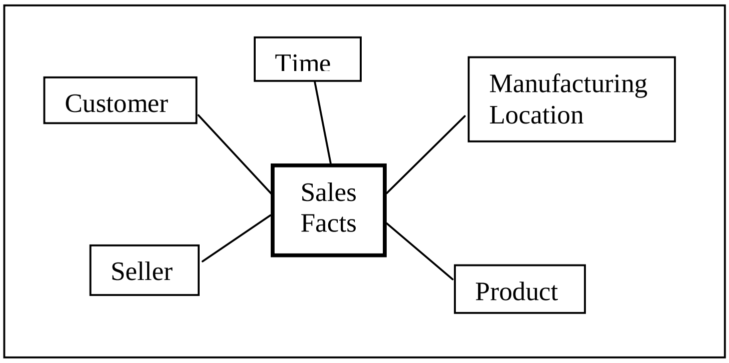


Figure 2.3

Other models that implement different design alternatives can be used. In [Kor99] they present a number of them, for example, *flat*, *terraced*, *star cluster*.

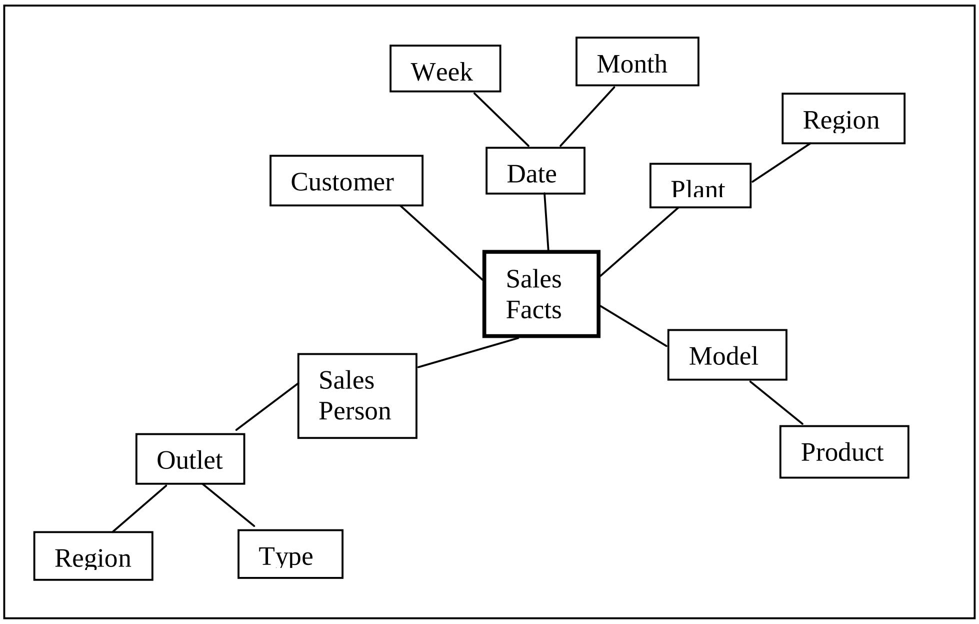


Figure 2.4

Practical design techniques and methods are proposed in [Kim96-1][Kim98][Kim96-3][Bal98], following mainly a star-schema approach. In [Ada98], authors also present concrete solutions for different target business. In [Sil97], they present DW models in a pattern-oriented approach, and propose techniques for converting a corporate logical data model into the DW model. In [Kor99] authors present a method for developing dimensional models from traditional Entity Relationship models.

In [Theo99-1] they focus on DW design, following the approach of *materialised views*. They address the problem of selecting a set of views to materialise in a DW taking into account: (i) the space allocated for materialisation, (ii) the ability of answering a set of queries (defined against the source relations) using exclusively these views, and (iii) the combined query evaluation and view maintenance cost. In this proposal they define a graph based on states and state transitions. They define a state as a set of views plus a set of queries, containing an associated cost. Transitions are generated when views or queries are changed. They demonstrate that there is always a path from an initial state to the minimal cost state. Some other publications about this approach are [Theo99-2][Lig99]. In [Theo99-3], working with the same model, they address the "Dynamic DW Design Problem", where basically, they determine which additional views have to be materialised when new queries have to be answered by the DW.

Our approach for DW design is not based on the materialisation of views. When using materialised views each desired relation of the schema must be able to be expressed in only one SQL query. Besides, we think that design process is easier and purer if it is done thinking only in the desired schema and not having to construct adequate SQL queries for obtaining the desired structures. In our work we clearly separate schema design from data loading, and we concentrate on schema design. According to our approach a DW schema can be designed transforming the source schema through a set of primitives and not depending on SQL expressiveness. Once the DW schema is designed, loading processes can be constructed.

Our goal with respect to the set of primitives we designed is that they embed DW design techniques, covering all the possible basic transformations that may be necessary for obtaining a DW schema from a source schema. In order to achieve this goal we base on the existing bibliography about DW design practical techniques and methods.

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

The present work intends to abstract and structure DW design techniques and strategies in the schema transformation primitives.

4. Schema transformation

The use of schema transformation primitives is a classical conceptual tool in Databases area. In [Bat92], design primitives and strategies are presented as the building blocks of conceptual design methodologies. In [Hai91], they analyse the concept of schema transformation and generalise many of the proposed transformations in a conceptual schema design context. In [Sta90], database schema transformations are used and automated to perform schema evolution and reorganisation.

Our work proposes schema transformation primitives for relational DW design.

5. Schema evolution

A big amount of work has been done on schema evolution. We present in this section only the concepts that are the most relevant and applicable to our case.

The proposals existing in the consulted bibliography about schema evolution always deal with Object Oriented (OO) Databases. In order to apply this knowledge to our context, we will have to do a mapping of the presented concepts and techniques, to Relational Databases.

In general (e.g. [Zic91][Fer96][Ska86]) we found that two main aspects are taken into account with respect to the state of a database after schema evolution: (i) **structural consistency** and (ii) **behavioural consistency**. In [Fer96] these concepts are defined in the following manner. Structural consistency is the consistency between the database and the schema. Behavioural consistency is to keep the consistency of the application programs that existed before evolution. The different authors concentrate in maintaining these properties.

On the other hand, we found two approaches for managing schema evolution: (a) **Adaptational approach** [Fer93][Fer94][Fer95] and (b) **Versioning approach** [Ska86][Fer96][Lau96][Lau97][Ngu89]. In the adaptational approach, when the schema is modified the state of the schema before the change is lost and the final result of evolution is an only one schema with the new structure. The existing instances have to be adapted to the new schema and the application programs that run over the database before the changes, also have to be adapted. In the versioning approach, modifications to the schema are not applied directly on the existing schema. Instead, a new version of the schema is created. In this case the existing instances do not necessary have to be transformed to satisfy the new schema. Besides, the application programs will continue running with the same behaviour over the previous version of the database; they neither have to be adapted to the new schema.

When adaptational approach is chosen for managing schema evolution, another dilemma comes up: how to manage the unavoidable update of the existing data. This problem is addressed in [Fer93][Fer94][Fer95]. There are mainly two options: (1) immediate updates and (2) lazy (deferred) updates. In the first case, data is updated immediately after a modification is done to the schema. In the second, data is updated at the moment it is used. These two strategies are intended to have the same final result: the database reaches a consistent state with respect to the new schema. For the database updates the designer has to provide the conversion functions. Depending on the complexity of these functions each strategy can be better applied or not. Various algorithms are proposed for implementing lazy updates [Fer94]. They address the problems of complex conversion functions and cycles that can be generated in the execution of the updates. In [Fer95] benchmarks are performed in order to compare the two possible strategies considering different contexts.

When versioning approach is used [Lau96][Lau97][Fer96] a list of schema versions with a relationship "is-derived-from" is managed. Only the last version of the list can be modified; the other ones are "frozen". This mechanism allows having different schema states, which gives the possibility to go back to a previous state if some update led to an unexpected result. In addition, with this mechanism existing applications can continue working over previous versions. The problem that has to be solved in this case is how to share data between the different schema versions. For this, three main concepts are defined and managed: (i) Instance Access Scope (IAS), (ii) conversion functions, and (iii) propagation flags. (i) The IAS of a schema version is the part of the database that is visible through this version. The IAS contains instances that were created by this version and instances that were propagated from other versions. (ii) Conversion functions are used to transform data to the new version of the schema. They are implemented at class level, and there are default conversion functions. In order to share instances between versions, two kind of conversion functions exist: forward conversion functions (f.c.f) and backward conversion functions (b.c.f). As an example, to *read* old data from the new version, you have to *read* and then *transform* (f.c.f.), to *write* old data from a new version you *transform* (b.c.f.) and then *write*. Many f.c.f or b.c.f can be composed for propagating data throughout chains of versions. (iii) The propagation flags are 4 flags that the designer must define when a new version is derived. With these flags he defines which parts of the superversion's[3] database will be shared by the subversion and what kind of operations the subversion will be able to apply over this database.

In [Fer96] an integration of the two approaches presented previously (adaptational and versioning) is proposed. Considering the fact that using adaptational approach we might invalidate applications, which are already running on top of the database, and versioning approach might burden a large overhead on the system, they propose a new solution. The idea is to apply the schema updates (adaptational) for certain cases, where applications are not corrupted by the updates, and to create a new schema version for the other cases. They categorise the schema changes into extending and modifying ones, and determine which approach should be used for each case.

In our proposal we extract some techniques from this existing work, and we adapt, combine and apply them for the resolution of our problem.

There is in the literature much work about taxonomies for evolution and the effects of each operation on the schema and its instances [Ban87][Ska86][Zic91]. However, this work is not so useful for us because it is specific for OO databases.

[3] Superversion and subversion are the predecessor and successor in the derivation list of versions, respectively.

Finally, in [Ban87] they define a set of invariants of an OO schema, which must be preserved throughout the schema changes, and they define a set of rules that state how to preserve the invariants for each schema change. In our work we use a very similar approach for preserving consistency in the DW schema, when it is constructed (Chapter 3, Sections: 3, 5) as well as when it is modified (Chapter 4, Section 3.3).

6. DW schema evolution

The work we have found in latest publications about DW evolution shows that this is an interesting and important problem to address, but it has not been very much explored yet.

The EVE (Evolvable View Environments) project [Nic98][Run97] studies the problem of how to maintain a DW under data and schema changes. When there are source schema changes they rewrite view definitions adapting all affected materialised views. This is called View Sinchronization. A master thesis related to this project [Zha99] proposes solutions to concurrent updates problems, reusing the proposal of EVE for View Synchronization and integrating it with other solutions for View Maintenance.

In addition, we have found some work about multidimensional (MD) schema evolution and some other work that concentrates on the impact DW evolution has on DW quality.

In the first [Bla99-1][Bla99-2], they define a conceptual model for MD schemas and instances and present a list of evolution operations including the effects they have on the MD schema. They consider only evolution that occurs over the DW schema as a consequence of user requirements changes.

In the second [Qui99][Vas99], they extend a process model for DWs they had previously defined, with the capability of representing DW evolution processes. In their approach, DW evolution processes that are executed on the DW are stored in the metadata repository, and then information can be extracted for analysing the impact that evolution operations had on the DW. They present a list of DW evolution operations with the quality factors and schema structures they affect.

7. Conclusion

In this chapter we presented the state of the art in the areas that are related to our work.

With respect to DW in general, the most focused problems are Data Integration, Extraction and Transformation, DW Maintenance and DW Design. For conceptual design Multidimensional data model is used, while logical design can also be done upon a Relational model. Materialised views is the most used approach for DW management.

In the area of DW Design we find works about how to select the views to materialise for a DW. The other bibliography we consulted presented techniques and strategies for relational DW design.

Schema Transformation is used in some proposals as a tool for constructing or evolving database schemas.

In the area of Schema Evolution there are different approaches for solving the problem of changes in a schema. The most relevant ones are: Adaptational and Versioning approach. With respect to DW Schema Evolution we have found work about multidimensional schema evolution and work that concentrates on the impact DW evolution has on DW quality.

CHAPTER 3. Data Warehouse logical design

1. Introduction

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.

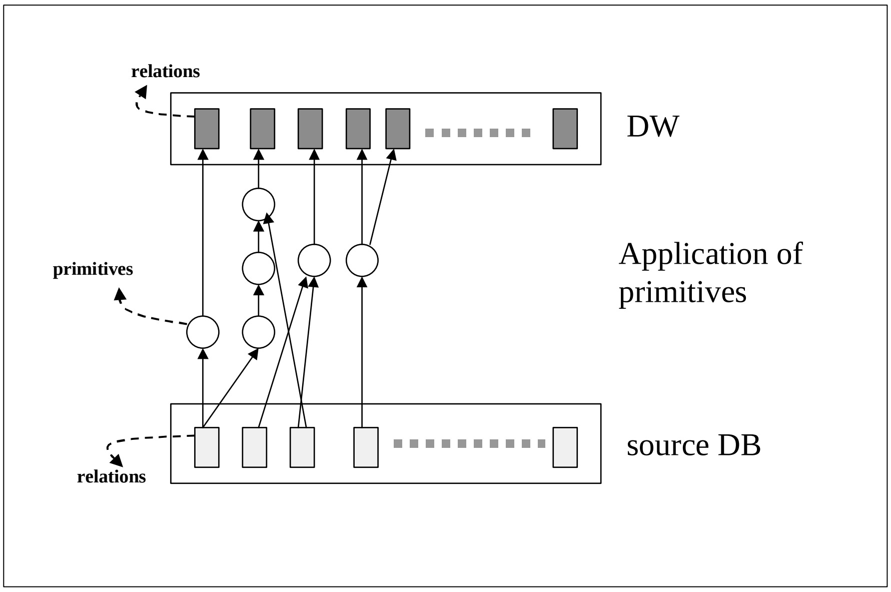


Figure 3.1

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[4] 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.

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 Section 2 we present some basic definitions about the model we use for the specification of the primitives. In Section 3 we define a set of DW schema invariants for DW schema consistency. In Section 4 we present the schema transformation primitives. In Section 5 we propose a set of consistency rules for application of primitives, and in Section 6 we propose a set of design strategies, which address the most common problems that appear when designing a DW. In Section 7 we show the format and specification of the trace that is generated when DW design is done through the primitives. Section 8 is the conclusion of this chapter. In Appendix 1 we present an example of a complete design process through primitive applications.

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

2. 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[5] 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.
We define a function $f_H : Rel_H$ -> Rel , which, given a historical relation, returns the corresponding current relation.

These sets verify the following properties:

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

Attribute sets:

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

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

$Att_M(R)$ – 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: $X \cap Y \neq \varnothing$

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 = $\{A_1, \ldots., A_n\} \wedge$

$A_1 < A_2 < \ldots. < A_n$, 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

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.

Primitive 4.1. VERSION DIGITS

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

Primitive 4.2. KEY EXTENSION

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

Primitive 5. 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.

Primitive 6. 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.

Primitive 6.1. 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.

Primitive 14. 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.

Primitive 1. IDENTITY

Description:

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

Input:

- source schema : R ∈ *Rel*
- source instance : r

Resulting schema:

- R’ ∈ *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) $\in$ *Rel*
- $X \subset \{ A_1, ..., A_n \} \wedge X \subset Att_D(R)$
- source instance : r

Resulting schema:

- R' (A'_1, ..., A'_m) $\in$ *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) / $\exists X \subset \{ A_1, ..., A_n \} \wedge X \in Att_K(R)$
- 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 $\in$ Att_D $\wedge$ if key then XT $\in$ $Att_K(R)$

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 : $R (A_1, ..., A_n) \in \mathit{Rel}$ / $X \in \mathit{Att}_{FK}(R)$
- 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
- $S \in \mathit{Rel}$ / $\mathit{Att}(S) = X \cup Y$, auxiliary relation that contains the correspondence between the old key and the new key
- Source instance : r, s

Resulting schema:

- $R' \in \mathit{Rel}$ / $\mathit{Att}(R') = Y \text{ U } (\{ A_1, ..., A_n \} - X) \wedge Y = \mathit{Att}_{FK}(R',R_1) \wedge ....$
  $\wedge Y = \mathit{Att}_{FK}(R',R_m)$

Generated instance:

- r' = select $Y \cup (\{A_1, ..., A_n\} - X)$
  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.

rels \ tuples	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**.

Primitive 6.1. **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) $\in$ *Rel*
- f (A_{i1}, ..., A_{im}) / { A_{i1}, ..., A_{im} } $\subseteq$ { A_1, ..., A_n } , where f is a user-defined function
- source instance : r

Resulting schema:

- R' (A_1, ..., A_n, A_{n+1}) $\in$ *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

Result:

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

Resulting schema:

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

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

♦

Primitive 11. 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 $\in$ $Att_K(R)$

Primitive 11.1. 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 $R_1 (XY) \in \mathit{Rel}_D$ / $X \in \mathit{Att}_K(R_1)$
- if $Z \neq \varnothing$ then $R_2 (XZ) \in \mathit{Rel}_D$ / $X \in \mathit{Att}_K(R_2)$
- if $W \neq \varnothing$ then $R_3 (XW) \in \mathit{Rel}_D$ / $X \in \mathit{Att}_K(R_3)$

Generated instance:

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

Primitive 11.2. 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:

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

Note: 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:

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

Primitive 12. **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 $\in$ $Att_D(R_i)$, 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 $Att(Ri) \cap \{J_1, ...., J_m\}$
                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 / $\forall$ j:1..n, $J_i \notin Att(R_j)$ 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 $r'_i$
```

Primitive 12.3. 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) ∩ {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 / ∀ j:1..n, J_i ∉ 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 $\{Att(R_1) - Y_1\} \cup \{Att(R_2) - Y_2\}$
       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 / $Att_{FK}(R', S) = \emptyset$ $\wedge$ $\exists$ 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[6] **–**

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 $\in$ $Att(R')$

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.

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

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:

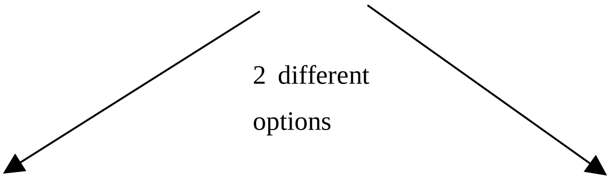
5.1) Vertically partition the relation, according to attribute values stability, through **Vertical Partition** primitive (P11.1).

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

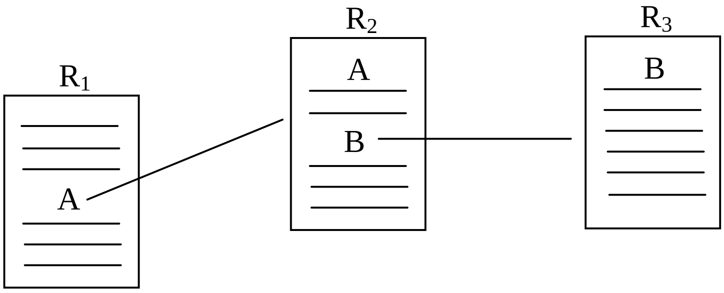
1- Normalised

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.

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

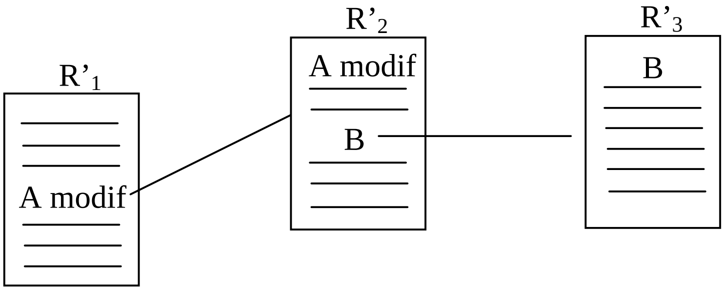
Possible strategies:



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[7], 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.

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.



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

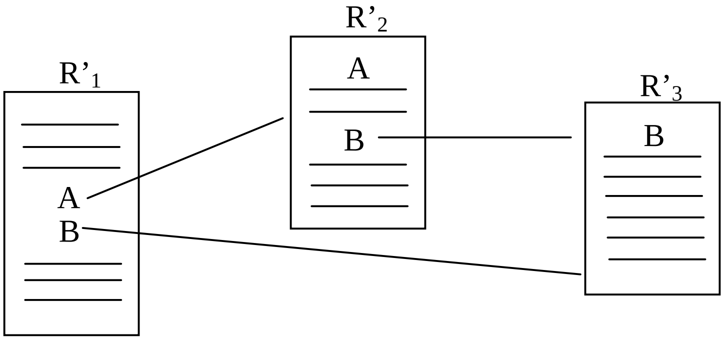
Steps:

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.

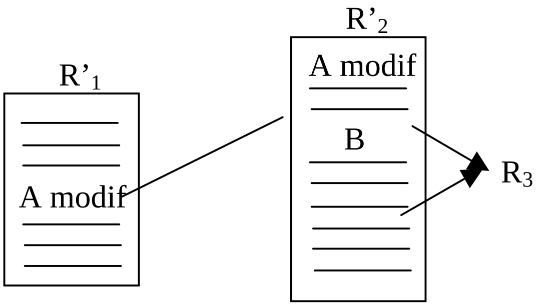


Steps:

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:

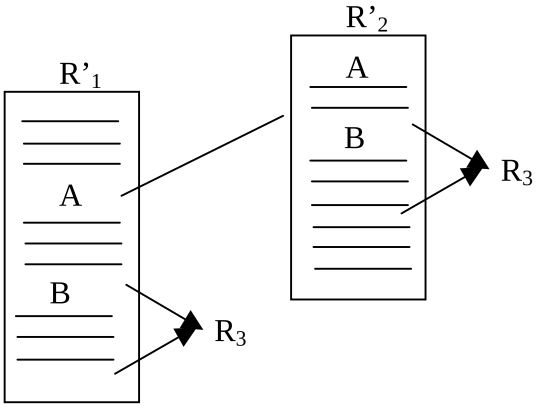
1) Apply to R_2 the alternatives **S1)**, **S2)** or **S4)** of the Versioning strategies presented earlier.

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

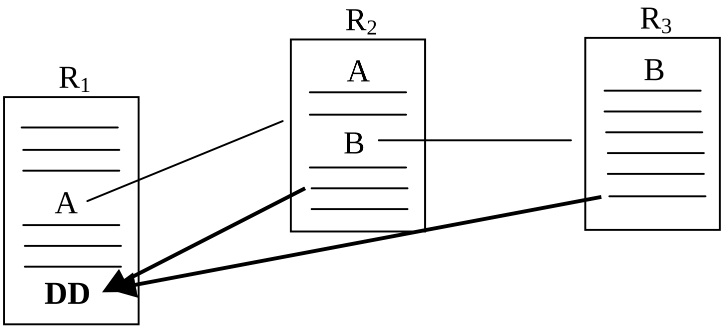
♦

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

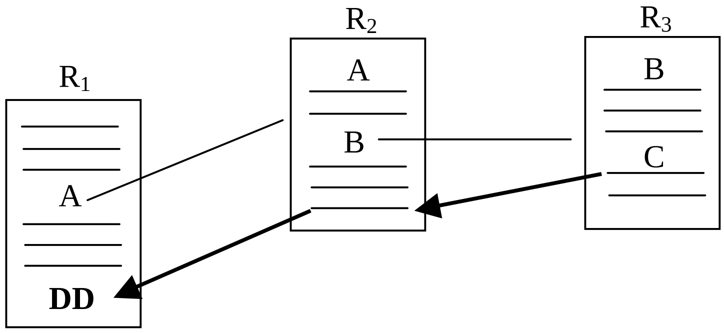
PLAN	**QUOTE**	**QUOTE_ VALUE**
100	1	500
100	2	495
100	3	490
101	1	350

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. 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.

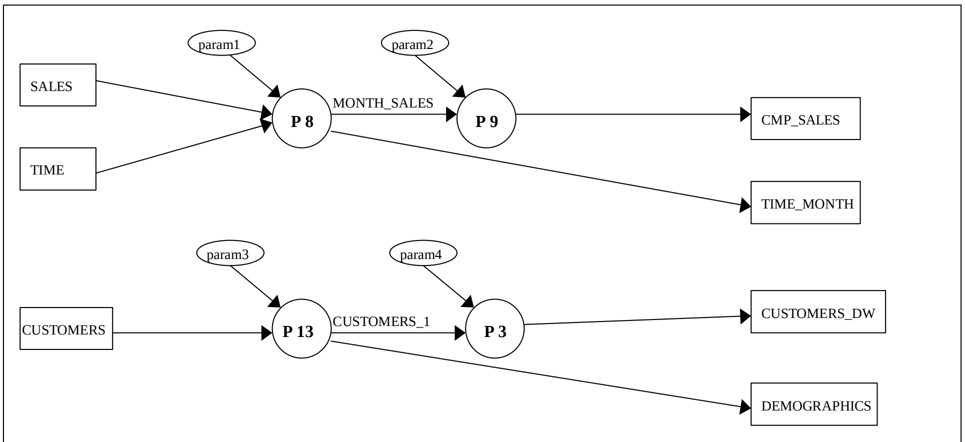
Edges:

- 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

♦

8. Conclusion

In this chapter we presented transformation primitives and guidelines for designing a DW schema starting from a source schema and taking into account DW requirements. The underlying data model is the Relational model, classifying the elements according to DW concepts.

Our proposal is based on the transformation of the source schema into a schema whose structures are more suitable for DW requirements. The proposed approach is based on a set of transformation primitives, which are applied to the source sub-schemas and generate new sub-schemas. The primitives are high-level operations that allow using different design techniques and should be applied with clear criteria. We also provided the designer some help in this direction. We defined a set of DW schema Invariants that are consistency properties, a set of Consistency Rules that ensure the satisfaction of the Invariants after primitives application, and some Design Strategies that act as guidelines for solving frequent DW design problems through application of primitives. In addition, we defined and specified the Transformation Trace, which is a tool that allows obtaining all the transformations that were applied to a schema element.

CHAPTER 4. Source schema evolution

In the previous chapter we presented a mechanism for designing a DW starting from a source database, through application of transformations. Once the design process has finished and the DW is completely generated, this DW remains linked to the source database through the trace that has been generated during the design. The link between DW and source database can be exploited at least for: (i) generating data loading processes (from source database to DW) and (ii) propagating to the DW changes occurred to the source database schema.

In this chapter we address the problem of propagation to the DW of source schema evolution.

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.

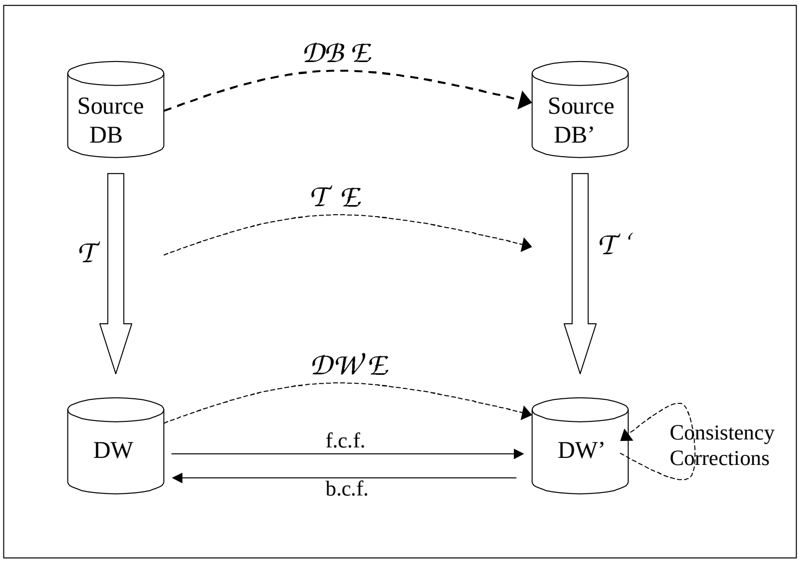


Figure 4.1

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.

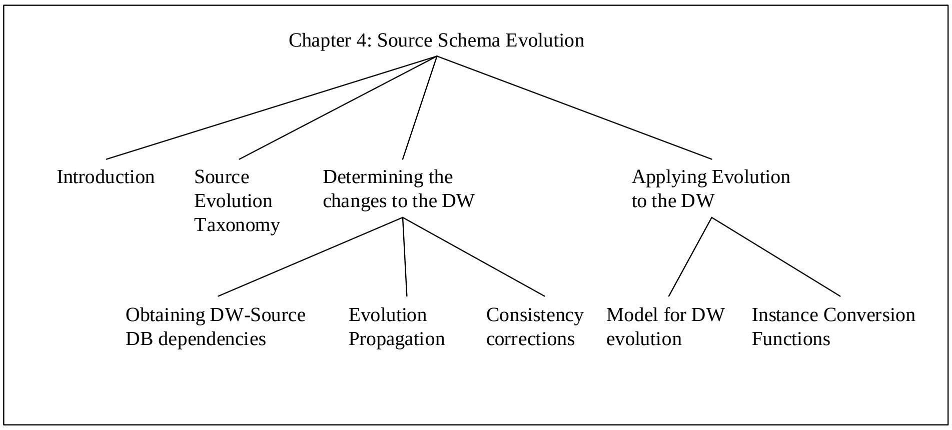

Chapter 4: Source Schema Evolution
Introduction
Source Evolution Taxonomy
Determining the changes to the DW
Applying Evolution to the DW
Obtaining DW-Source DB dependencies
Evolution Propagation
Consistency corrections
Model for DW evolution
Instance Conversion Functions

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.

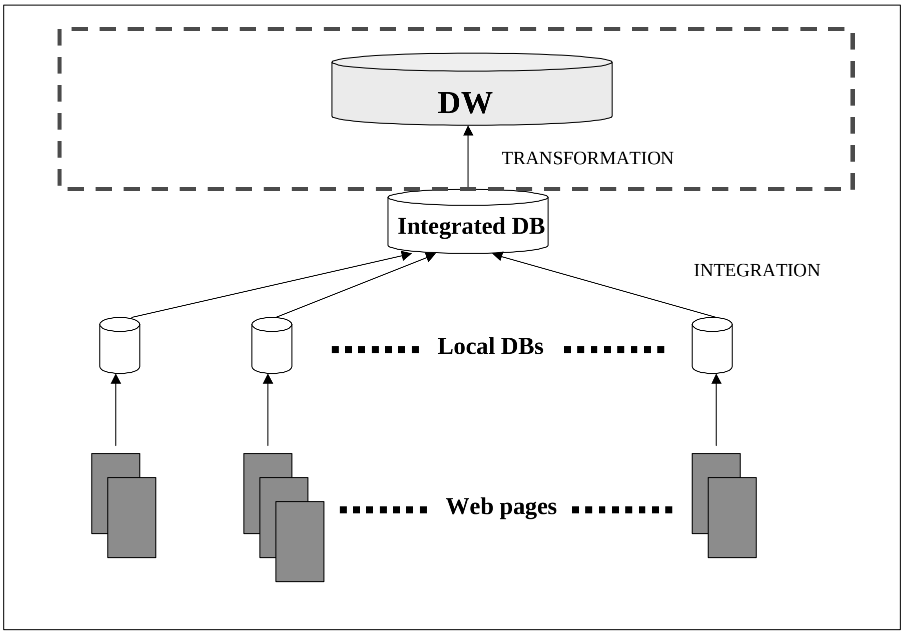


Figure 4.3

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 $\boldsymbol{T_D(R)}$

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:

$\boldsymbol{T_D(R)}$ ::= <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* $\Leftrightarrow$ $Att(n_2) = \mathrm{bop}(\, Att(n_1)\,)$,
bop $\in$ *Basic_Operations*,
- Let n_1, n_2 / n_1, $n_2 \in$ *Nodes*, $\exists$ $p(n_1, n_2) \in$ *Paths* $\Leftrightarrow$
$\exists$ $e(n_1, m_1), e(m_1, m_2), e(m_2, m_3), \ldots, e(m_N, n_2) \in$ *Edges*

♦

The general format of the graph is as follows:

$$\text{<attribute1>} \xrightarrow{\text{<op1>}} \text{<attribute2>} \xrightarrow{\text{<op2>}} \ldots\ldots\ldots\ldots \xrightarrow{\text{<op3>}} \text{<attributen>}$$

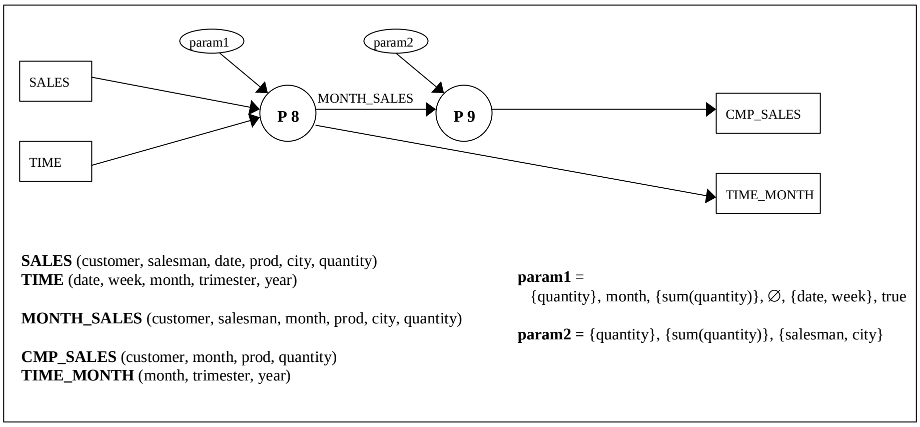
We illustrate the proposed mechanisms through an example.

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.

The trace of SALES:

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



SALES (customer, salesman, date, prod, city, quantity)
TIME (date, week, month, trimester, year)

MONTH_SALES (customer, salesman, month, prod, city, quantity)

CMP_SALES (customer, month, prod, quantity)
TIME_MONTH (month, trimester, year)

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

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

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.

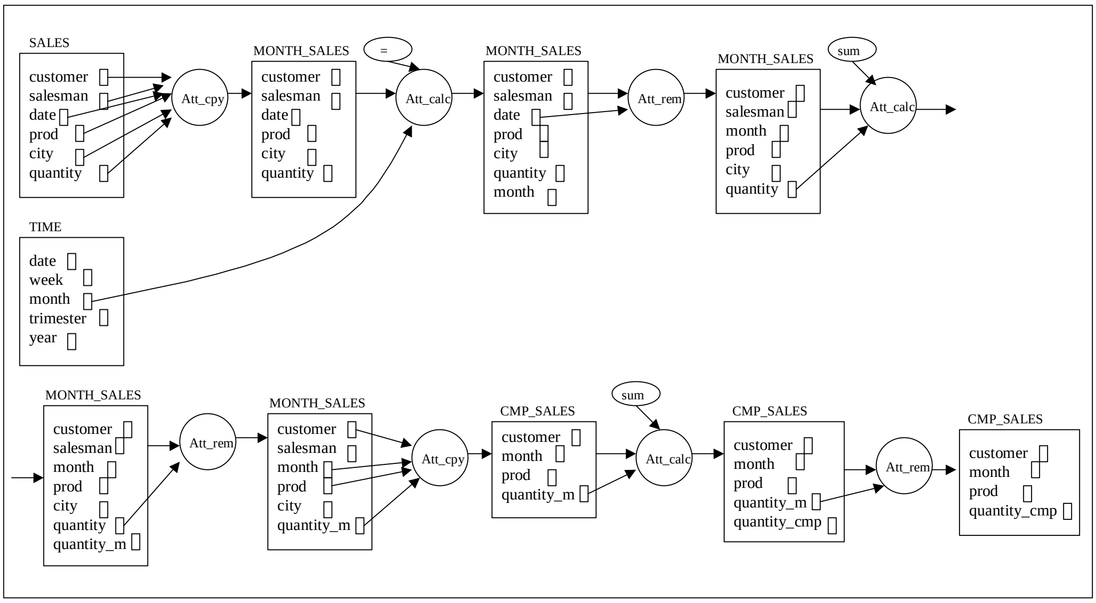


Figure 4.7

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:

$$\begin{tikzcd}
\text{SALES.quantity} \arrow[r, "\text{Att\_cpy}"] & \text{MONTH\_SALES.quantity} \arrow[r, "\text{Att\_calc}", "\text{sum}"'] \arrow[d, "\text{Att\_rem}"'] & \text{MONTH\_SALES.quantity\_m} \arrow[r] & {} \\
 & // & & \\
{} \arrow[r, "\text{Att\_cpy}"] & \text{CMP\_SALES.quantity\_m} \arrow[r, "\text{Att\_calc}", "\text{sum}"'] \arrow[d, "\text{Att\_rem}"'] & \text{CMP\_SALES.quantity\_cmp} & \\
 & // & &
\end{tikzcd}$$

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

$$\begin{tikzcd}
\text{SALES.customer} \arrow[r, "\text{Att\_cpy}"] & \text{MONTH\_SALES.customer} \arrow[r, "\text{Att\_cpy}"] & \text{CMP\_SALES.customer}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{SALES.date} \arrow[r, "\text{Att\_cpy}"] & \text{MONTH\_SALES.date} \arrow[r, "\text{Att\_calc}", "\text{req}"'] \arrow[d, "\text{Att\_rem}"'] & \text{MONTH\_SALES.month} \arrow[r] & {} \\
 & // & & \\
{} \arrow[r, "\text{Att\_cpy}"] & \text{CMP\_SALES.month} & &
\end{tikzcd}$$

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

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:

$$\begin{tikzcd}
\text{SALES.quantity} \arrow[r, "\text{Att\_cpy}"] & \text{MONTH\_SALES.quantity} \arrow[r, "\text{Att\_calc}", "\text{sum}"'] \arrow[d, "\text{Att\_rem}"'] & \text{MONTH\_SALES.quantity\_m} \arrow[r] & {} \\
 & // & &
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[r, "\text{Att\_cpy}"] & \text{CMP\_SALES.quantity\_m} \arrow[r, "\text{Att\_calc}", "\text{sum}"'] \arrow[d, "\text{Att\_rem}"'] & \text{CMP\_SALES.quantity\_cmp} \\
 & // &
\end{tikzcd}$$

Dependency expression of attribute SALES.quantity:

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

Trace of attribute SALES.customer:

$$\begin{tikzcd}
\text{SALES.customer} \arrow[r, "\text{Att\_cpy}"] & \text{MONTH\_SALES.customer} \arrow[r, "\text{Att\_cpy}"] & \text{CMP\_SALES.customer}
\end{tikzcd}$$

Dependency expressions of attribute SALES.customer:

CMP_SALES.customer = SALES.customer

Trace of attribute SALES.date:

$$\begin{tikzcd}
\text{SALES.date} \arrow[r, "\text{Att\_cpy}"] & \text{MONTH\_SALES.date} \arrow[r, "\text{Att\_calc}", "\text{req}"'] \arrow[d, "\text{Att\_rem}"'] & \text{MONTH\_SALES.month} \arrow[r] & {} \\
 & // & &
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[r, "\text{Att\_cpy}"] & \text{CMP\_SALES.month}
\end{tikzcd}$$

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) $\wedge$ **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) $\wedge$ 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 $\in$ $Att\_Name$
    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 $\in$ $DW_{SS}$'
                             - 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.

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.

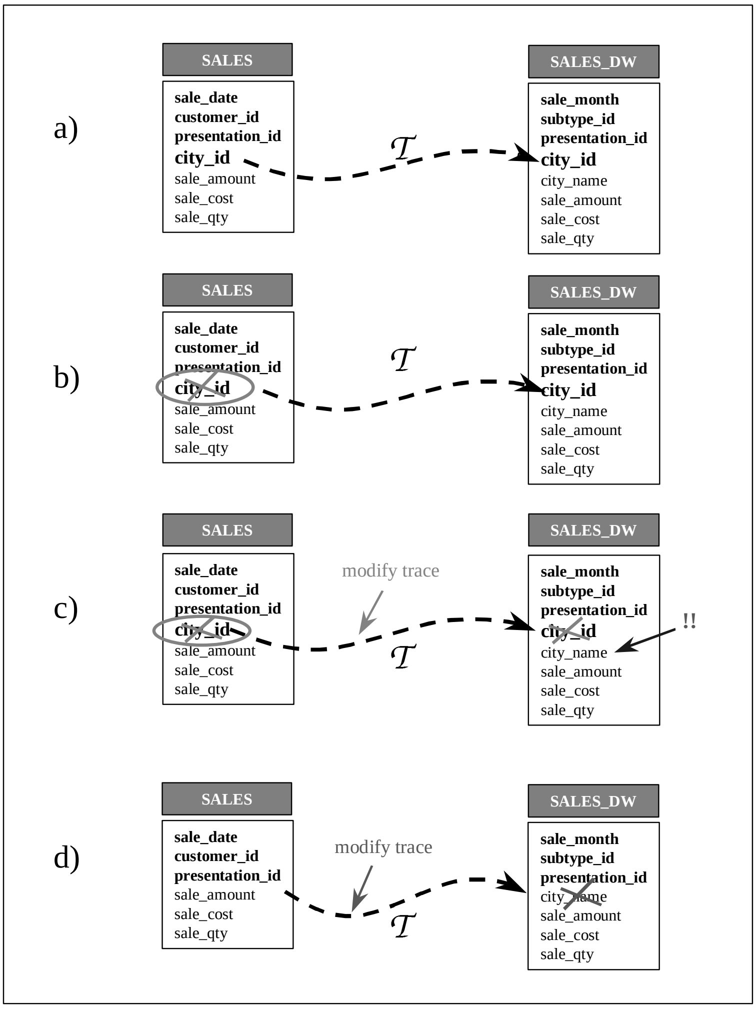


Figure 4.10

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 ∈ Rel_D

WHEN: ∃ R’ ∈ Rel_M / $Att_{FK}(R’, R) \neq \emptyset$

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

where X = { A / A ∈ *Att(R)* ∧ A ∈ *Att(R’)* }

R2 – Measure relations correction

ON APPLICATION OF: *Att_rem ({A}, R)* / R ∈ Rel_M

WHEN: ∃ S ∈ Rel_D / $Att_{FK}(R, S) = \emptyset$ ∧ ∃ B / B ∈ *Att(R)* ∧ B ∈ *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[8]

ON APPLICATION OF: *Att_add ({A}, R)*, obtaining A ∈ *Att(R)*

WHEN: ∃ R’ / R’ ∈ $Rel_H(R)$

APPLY: *Att_add ({A}, R’)*, obtaining A ∈ *Att(R’)*

[8] 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.

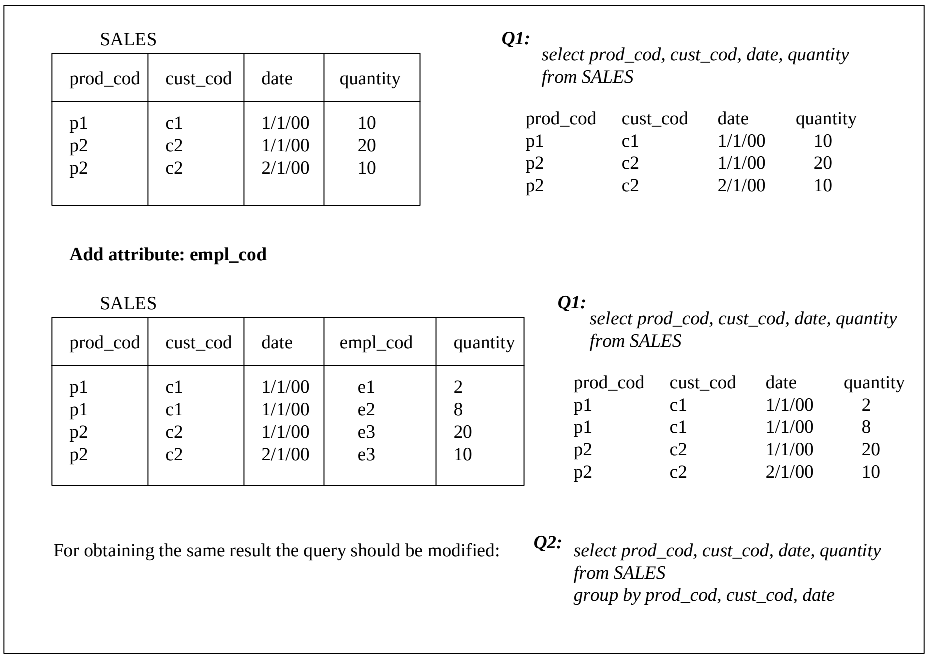
Some particular features of DWs:

- 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.



SALES

prod_cod	cust_cod	date	quantity
p1	c1	1/1/00	10
p2	c2	1/1/00	20
p2	c2	2/1/00	10

Q1:

```
select prod_cod, cust_cod, date, quantity
from SALES
```

```
prod_cod   cust_cod   date      quantity
p1         c1         1/1/00      10
p2         c2         1/1/00      20
p2         c2         2/1/00      10
```

Add attribute: empl_cod

SALES

prod_cod	cust_cod	date	empl_cod	quantity
p1	c1	1/1/00	e1	2
p1	c1	1/1/00	e2	8
p2	c2	1/1/00	e3	20
p2	c2	2/1/00	e3	10

Q1:

```
select prod_cod, cust_cod, date, quantity
from SALES
```

```
prod_cod   cust_cod   date      quantity
p1         c1         1/1/00      2
p1         c1         1/1/00      8
p2         c2         1/1/00      20
p2         c2         2/1/00      10
```

For obtaining the same result the query should be modified:

Q2:

```
select prod_cod, cust_cod, date, quantity
from SALES
group by prod_cod, cust_cod, date
```

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).

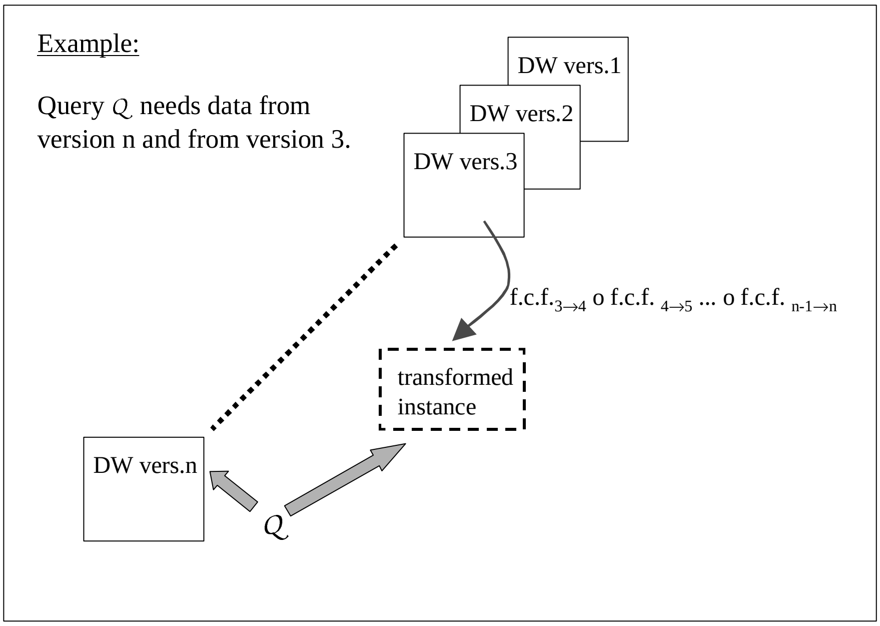

Example:
Query *Q* needs data from version n and from version 3.
DW vers.1
DW vers.2
DW vers.3
f.c.f.$_{3\to4}$ o f.c.f. $_{4\to5}$ ... o f.c.f. $_{n-1\to n}$
transformed instance
DW vers.n
Q

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.).

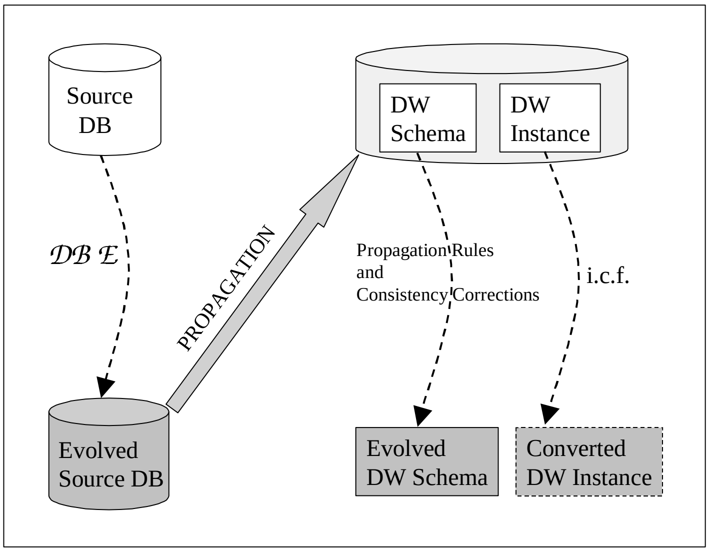


Figure 4.13

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 $\in$ *Rel_M*, A $\in$ *$Att_M(R)$*

- select Att(R) - A
  from R

d) from Dimension Relation

b1) descriptive attribute

i.c.f. 5: - R $\in$ *Rel_D*, A = *$Att_D(R)$*

- select Att(R) - A
  from R

b2) hierarchical attribute

i.c.f. 6: - R $\in$ *Rel_D*, A = *$Att_J(R)$*

- select Att(R) - A
  from R

3) Change key of a relation

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

- The instance must not be transformed

<u>Note:</u> 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 $\in$ *Rel*, {A} $\in$ *$Att_{FK}(R,R')$* old foreign key, B $\in$ *Att(R)* new foreign key

- The instance must not be transformed

<u>Note:</u> 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.

CHAPTER 5. Implementation

1. Introduction

1.1. Context

We have developed a prototype of a DW Design tool, called *DWDesigner*, that can be combined with other components conforming a CASE tool. These other components have been developed in the context of students' graduate projects and a demonstration that was presented in the ER'99 Conference [Per99]. The components are the following: (1) *CMDM (Conceptual Multidimensional Data Model)* [Pic99], (2) *A Repository Manager for a CASE tool* [Arz99], and (3) *From the conceptual schema to the logic schema of a DW* [Per00].

Figure 5.1 presents an overview of the architecture of the whole CASE environment.

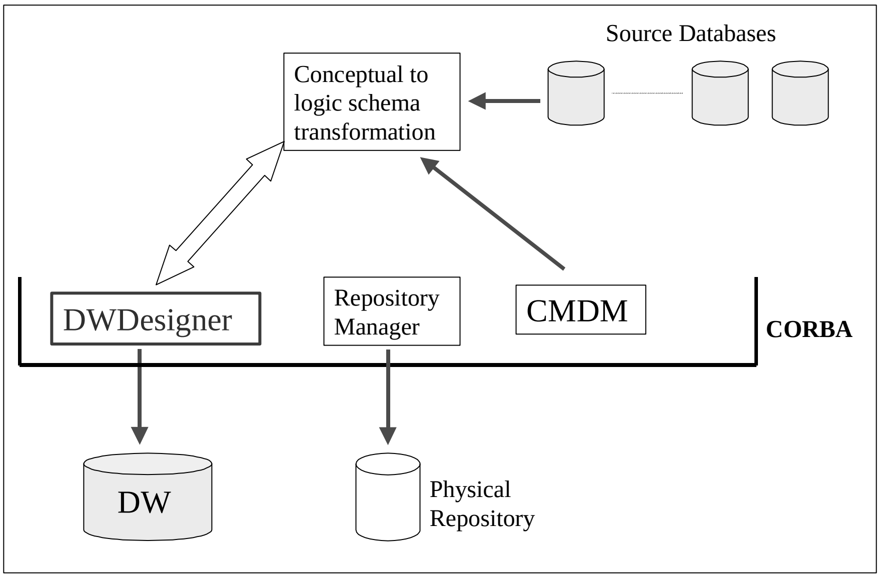


Figure 5.1

DWDesigner is a tool that implements the principal ideas of Chapter 3 of this thesis (DW logical design): transformation primitives, transformation trace, schema invariants and consistency rules. Implementation of DW evolution, whose ideas were presented in Chapter 4, is in course.

1.2. The prototype

The prototype is a DW Design tool with the following main characteristics:

- It allows the designer to design a DW schema starting form a source schema by means of Primitive applications.
- It generates a trace of the transformation applied.
- It provides invariants checking.
- It includes consistency rules triggering.
- It has a graphical user interface.
- It is extensible. Its modular design allows adding and removing primitives without modifying or re-compiling the existing code.

Most of this prototype was designed and programmed by a group of students in the context of their graduate project [Gar99], during 1999. The author of the present thesis co-directed[9] this graduate project and therefore participated in the analysis and design of the prototype. In addition, she developed alone (analysis, design and implementation) invariants checking and rules triggering functionalities, which had been left as future work in the mentioned graduate project.

Currently, we are directing a new graduate project that will implement DW evolution functionality for the existing tool.

In Section 2 we present a brief description of the prototype and in Section 3 we present the conclusions of this chapter.

2. Prototype description

In this section we pretend to give a descriptive view of the whole prototype (for a detailed description see [Gar99]) and give more detail about the modules we developed and how they integrate with the rest of the modules.

First we present a summarised analysis of the tool's functional features, then we present the most relevant aspects of the conceptual design, and by last we make some comments about the implementation.

[9] Together with Professor Alejandro Gutierrez.

2.1. Functional Features

The tool's main features were enumerated in Section 1.2 of this Chapter.

The tool is intended to be a DW design graphical environment. It should be useful to a DW designer who has a source database, some DW requirements, and wants to generate a DW schema that satisfies these two elements. With this tool the designer can apply different DW design criteria and techniques in order to obtain the target DW schema.

Invariants checking can be invoked at any moment in the design process. It allows checking the consistency of the schema that is being generated.

Consistency rules are triggered by the application of certain primitives to certain elements; when these applications put in danger the consistency of the schema that is being generated.

The graphical interface facilitates interaction with the source database elements, application of primitives and parameters selection, and visualisation of the design trace.

Figures 5.2, 5.3, 5.4, 5.5 and **5.6** show the interface of the tool.

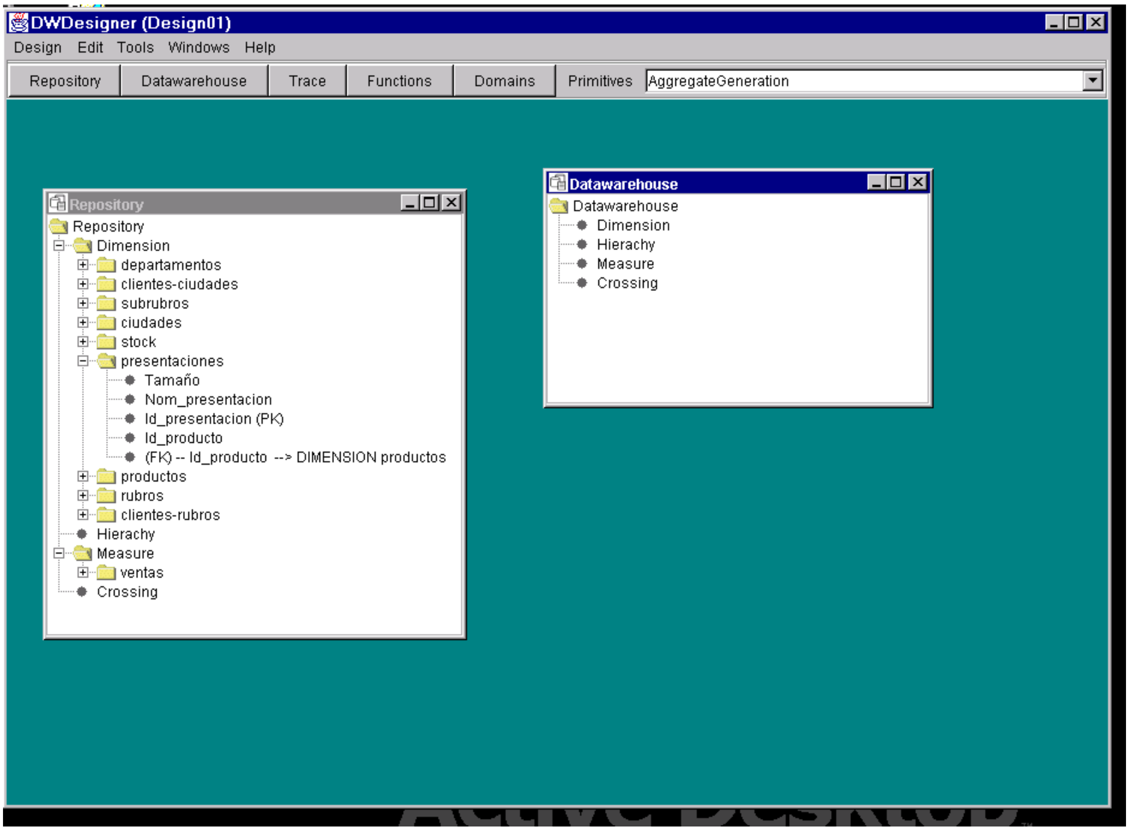
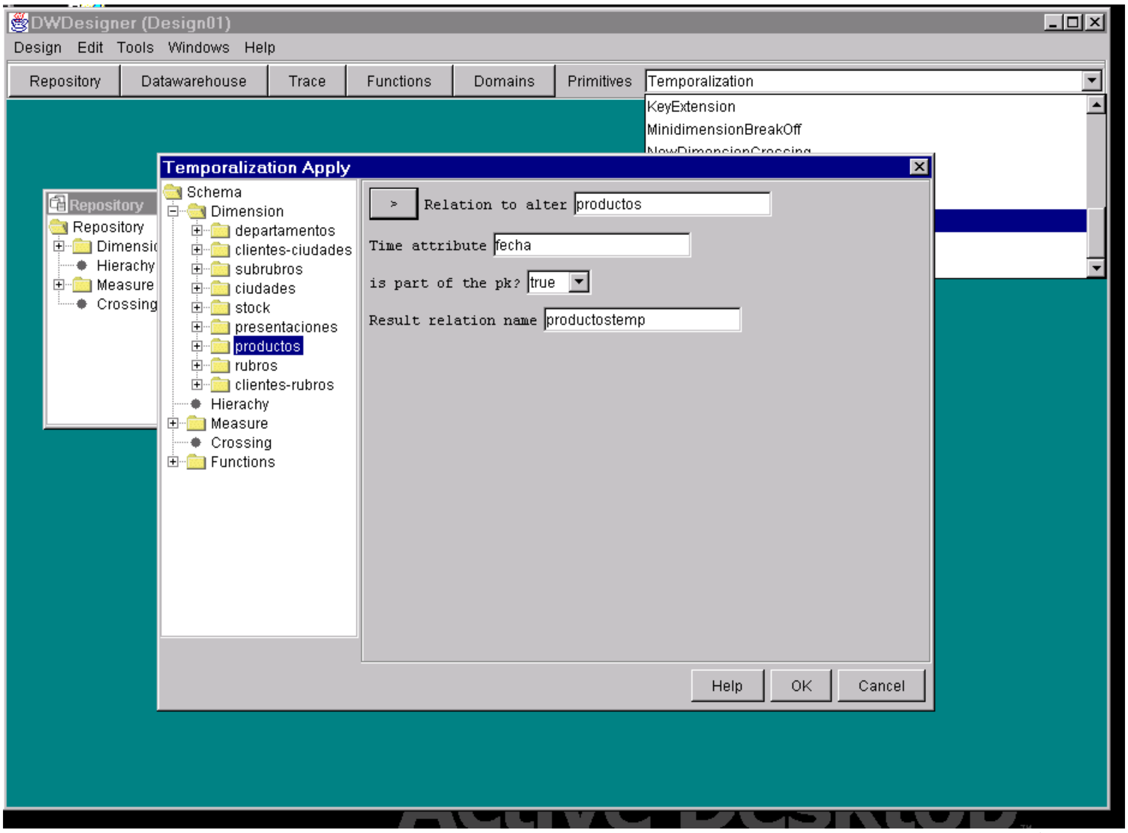


Figure 5.2



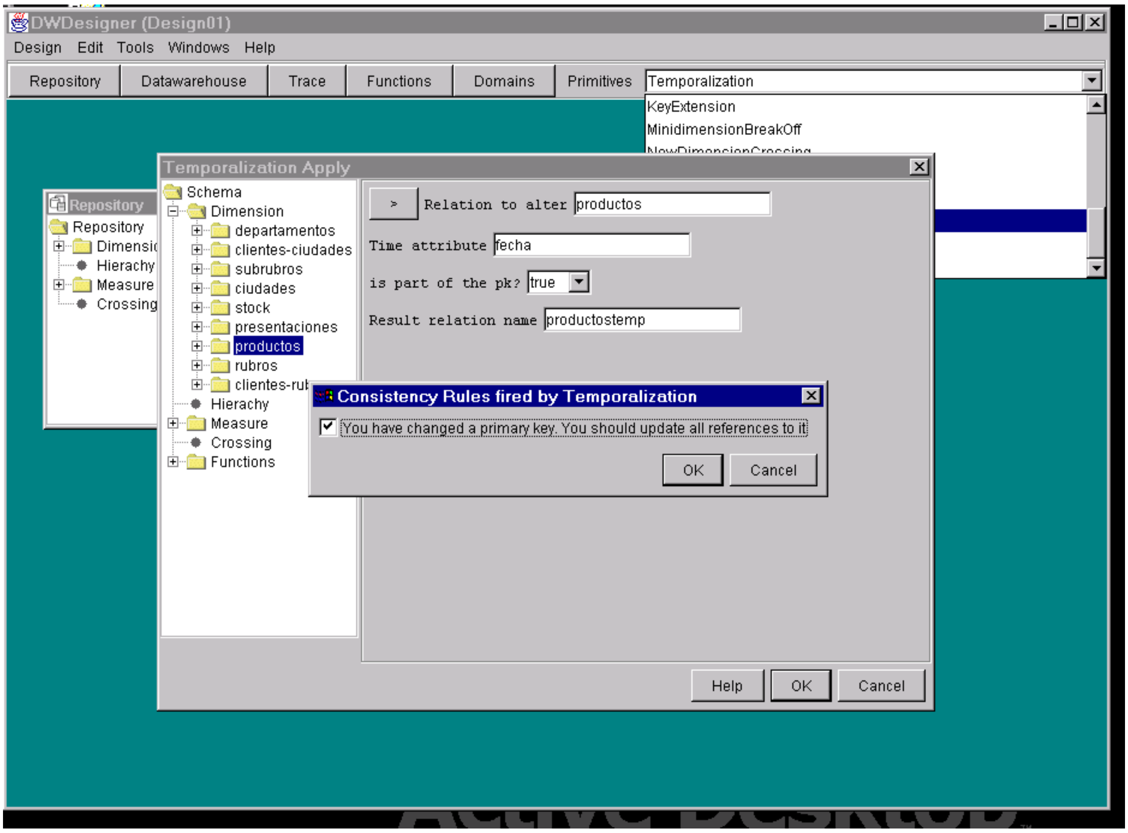
Relation to alter productos

Time attribute fecha

is part of the pk? true

Result relation name productostemp

Figure 5.3



Relation to alter productos

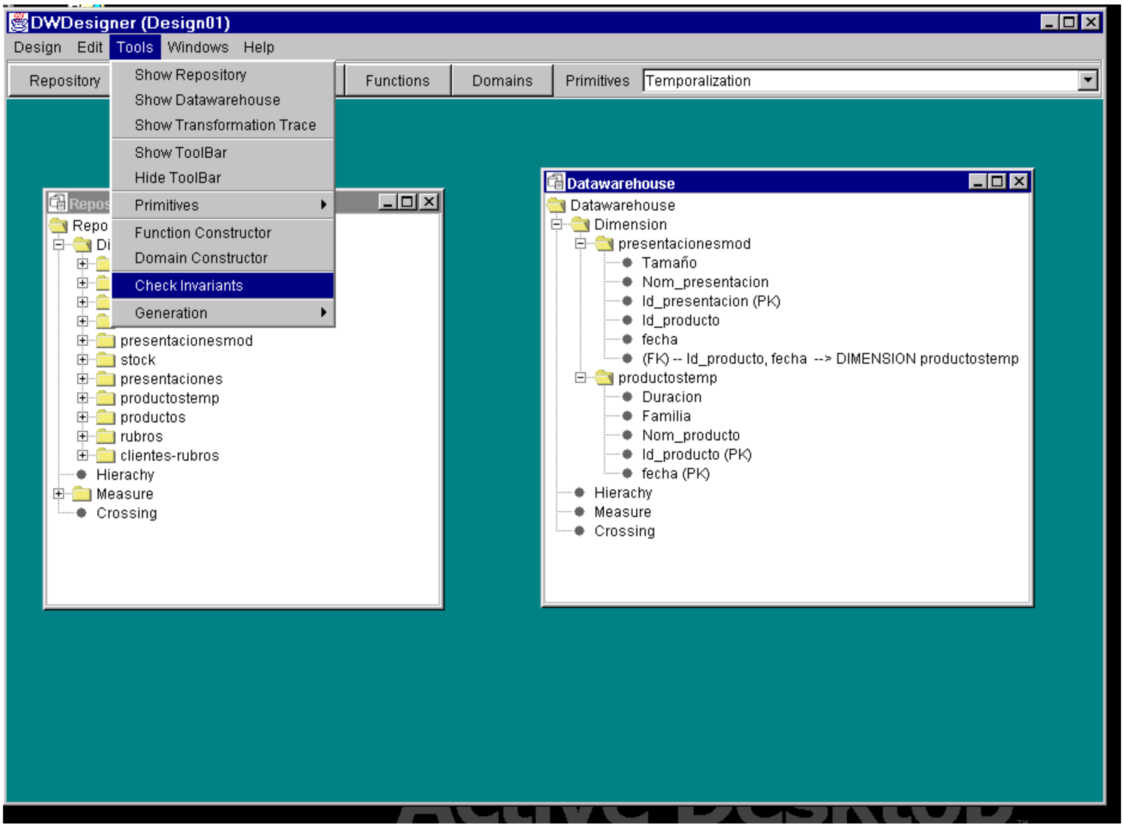
Time attribute fecha

is part of the pk? true

Result relation name productostemp

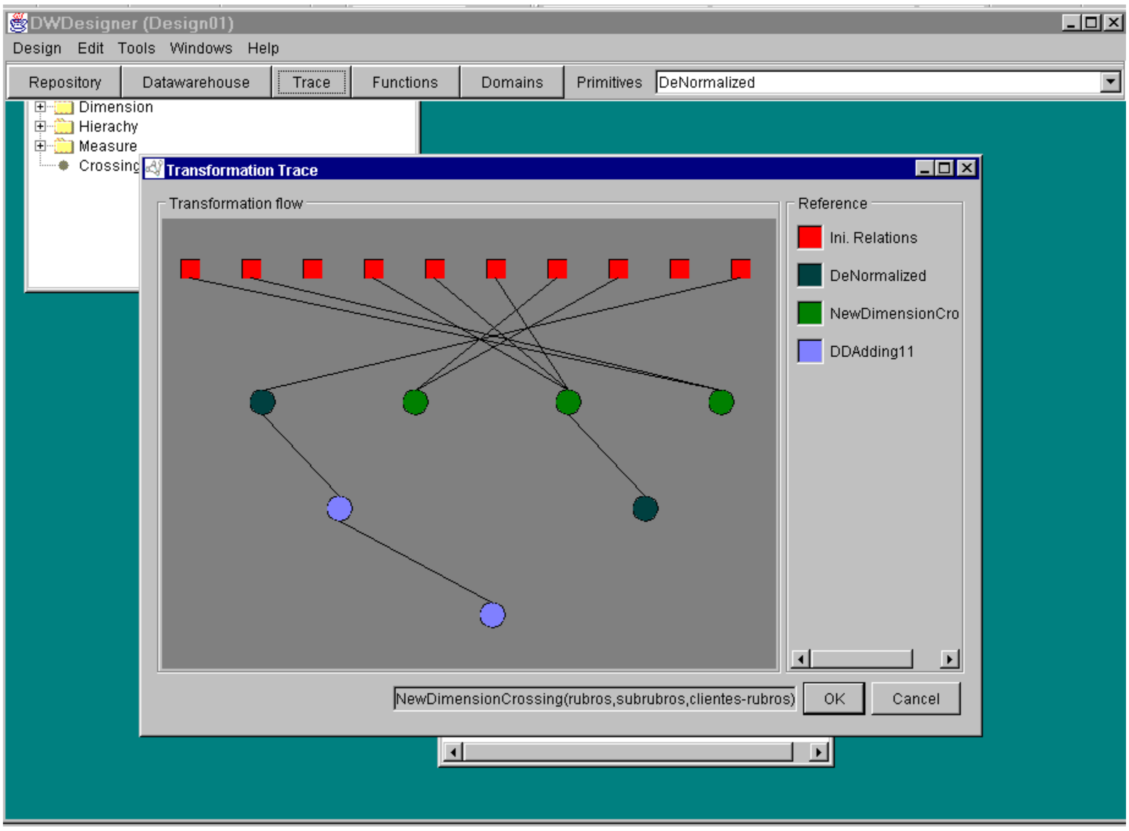
☑ You have changed a primary key. You should update all references to it

Figure 5.4



Primitives: Temporalization

Figure 5.5



Primitives: DeNormalized

NewDimensionCrossing(rubros,subrubros,clientes-rubros)

Figure 5.6

2.2. Conceptual design

For the design of the tool we applied object-oriented techniques. The design language we used was UML.

The core of the tool is the *Virtual Machine*. This is the most important layer of the system. The main components of the *Virtual Machine* are the following:

1) Tools for schema transformation

   This is the representation of: the transformation primitives, and two sets of schema elements (called Repository and Datawarehouse) that are the containers used during the transformation process.

2) Database concepts representation

   All schema elements, relations, attributes, keys, etc. must be represented in order to be manipulated by the user and the primitives.

3) Trace

   Each applied primitive, with its input and output relations, is recorded in the trace.

In the context of the present thesis, the students' work was extended with:

4) Invariants

   The invariants are properties that must be satisfied by the schema. Therefore we represented the invariants as a part of the schema representation.

5) Consistency Rules

   We represented the Rules as an independent entity, which is referenced and invoked from different places.

The class diagram of the DW design tool can be seen in Appendix 4.

2.3. Implementation

The prototype was implemented in JAVA language, using the Java Development Kit version 1.2.2 (JDK 1.2.2) and Borland's JBuilder version 2 as the development environment.

Implementation details of the tool, excepting the parts of Invariants and Rules, can be found in [Gar99]. In this Section we briefly explain how we implemented Invariants Checking and Rules Triggering.

Invariants Checking is an option of the tool's main menu that allows checking the consistency of the existing DW schema. The user has the possibility of choosing between a list of invariants. We implemented the procedures that perform the Invariants Checking as methods of the *General Schema* class. When the user press *OK* button, we call the methods of the *General Schema* class that correspond to the invariants selected by the user.

Rules are implemented as an abstract class *Rule* and a sub-class *RuleXX* for each existing rule (analogous to the primitives). We defined the *RuleDirectory*, which is a sequence containing the existing rules and is initialised at start. Also at this moment, some primitives initialise the *rules* attribute that is a set of rules (referencing to rules of the RuleDirectory). The rules that belong to a primitive's set of rules are the ones that must be triggered after the primitive application. When the user applies certain primitives, a dialog box appears showing him what rules should be applied in the form of check boxes. The rules that are checked by the user are applied automatically. The transformations that are made by the rules, are applications of primitives, therefore they have the same behaviour as any primitive application (e.g. they are reflected in the trace).

3. Conclusion

DWDesigner is a prototype of a DW Design Tool. We directed the development of this prototype and we developed a new part of it; Invariants Checking and Rule Triggering functionalities.

The developed tool implements the principal ideas of Chapter 3 of this thesis (DW logical design): transformation primitives, transformation trace, schema invariants and consistency rules. This tool offers a graphical user interface that allows the designer to apply primitives to a source schema, constructing a new schema, visualise the generated transformation trace, check schema invariants and apply consistency rules.

The tool can be connected with other modules, complementing each other. Altogether, they constitute a CASE tool for designing a DW that covers the stages of: conceptual modelling, derivation of a logical model, management of the logical model, and persistency of the design.

CHAPTER 6. Conclusion

This thesis addresses two main issues: DW design, and the repercussion of source schema evolution on the DW.

The obtained results consist of:

- a CASE tool for designing DWs by application of schema transformations
- techniques for repercussion of source schema evolution on the DW

1. DW design through schema transformations: techniques and CASE tool

The help tool for DW design 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.

The primitives, invariants and consistency rules were implemented in a DW design tool. The tool can be connected with other modules, complementing each other. Altogether, they constitute a CASE tool for designing a DW that covers the stages of: conceptual modelling, derivation of a logical model, management of the logical model, and persistency of the design.

Some other work that was done around our proposal for DW design is: experimentation with the primitives by applying them in real DW developments [Abe98], and presentation in ER’99 conference of a poster and demonstration containing it [Per99].

2. Repercussion of source schema evolution on the DW

The solution we propose for the problem of source schema evolution is applicable to a DW that was generated by application of the primitives, and uses the results obtained in the proposal for DW design. We propose a mechanism for repercuting source schema changes to the DW, which basically consists on deducing which changes have to be made to the DW and applying them.

For deducing the changes to be applied to the DW, we provide the following: (i) a taxonomy of source schema changes, (ii) a mechanism for obtaining dependencies between source and DW schema elements, (iii) a set of *propagation rules*, and (iv) a set of *correction rules*. For obtaining the dependencies we propose a way to process the trace so that more detailed traces are derived. The propagation rules state which changes must be performed on the DW and on the trace, depending on the elements' dependencies and the occurred source change. The correction rules assure consistency of the modified DW, being based on the schema invariants previously proposed in this work.

For applying the changes to the DW we propose a strategy that is based on the Versioning Model for object oriented schemas (presented in Chapter 2, Section 5). In order to make this choice, we analyse the features of DWs and the applicability of the existing models to DW evolution. We also propose some *instance conversion functions* that are the conversions of the instance of a schema version into an instance of another schema version. For stating these conversions we had to determine another taxonomy, a DW evolution taxonomy.

It is important to note that for solving the evolution problem we define 2 different taxonomies. First, we need the source evolution taxonomy, which is defined taking into account the context of the project [CSI99]. Second, we define the DW evolution taxonomy. In this case, the changes of the taxonomy are the ones that may be generated by the application of the propagation rules.

3. Ongoing work

At the moment, source schema evolution management is being implemented in the context of a graduate project. This will be an extension of the implemented DW design tool.

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].

4. Future work

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.

  By last, we think that the primitives could be complemented with the *basic operations* (proposed in this thesis for decomposing the primitives) at the moment of design, in case it is necessary. In addition, we think that considering the basic operations, completeness could be formally demonstrated, following these ideas: (a) In our context, a schema consists of relations, attributes, and definitions of keys and foreign-keys. (b) In the basic operations set exists an operation for adding a relation (Rel_add), an operation for adding an attribute (Att_add), and operation for defining a key (Key_add) and an operation for defining a foreign key (FKey_add). (c) *(a)* and *(b)* lead us to think that any schema can be constructed through application of basic operations of this set.

- 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 proposed 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 that was not addressed in this work.

Appendices

1. Appendix 1 – 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.

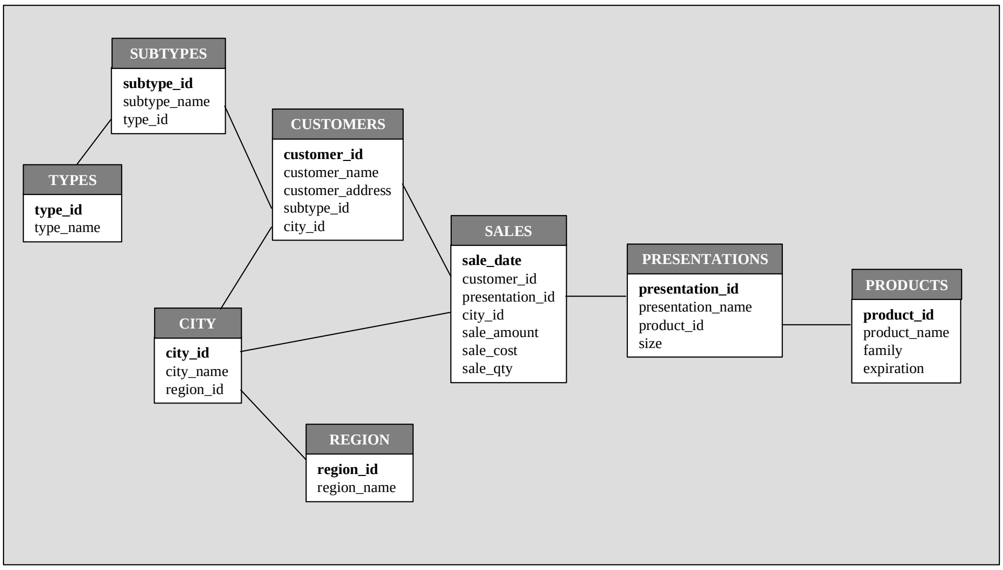


Figure 7: The source database schema

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[10], 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.

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

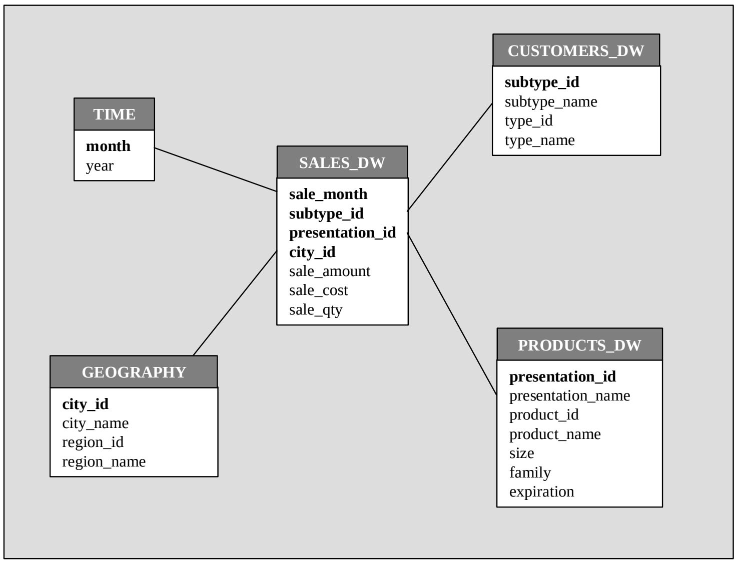


Figure 8: The target logical DW schema

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**.

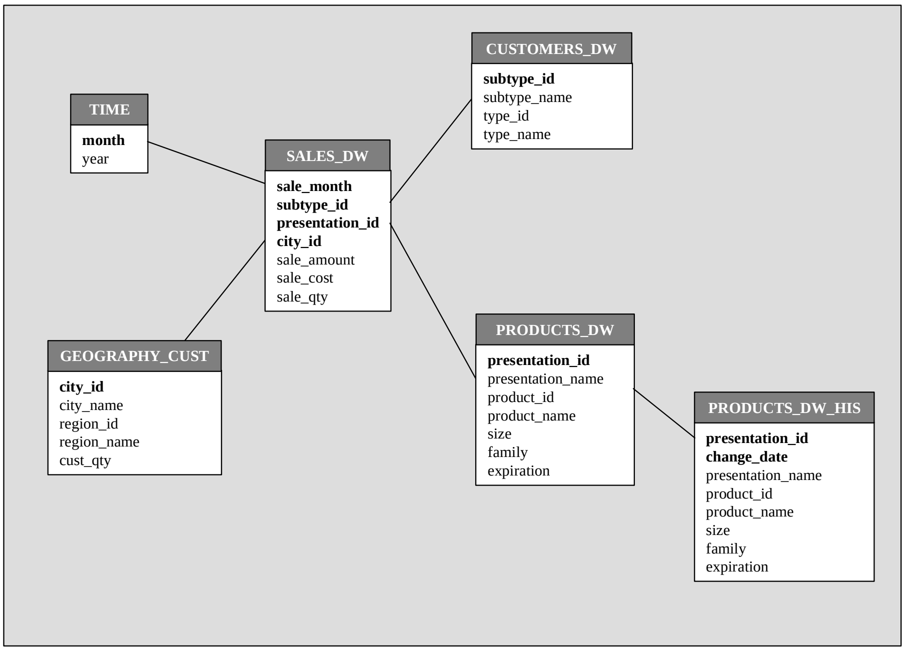


Figure 9: The obtained DW schema

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

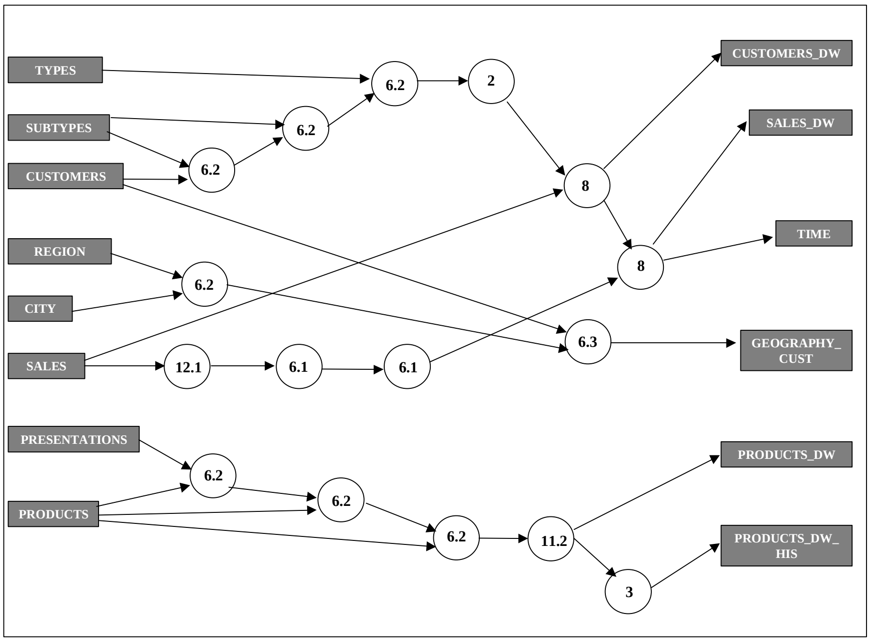


Figure 10: The generated trace

2. Appendix 2 – The Basic Operations

Operations over the context:

- **Rel_add** (R)

  Arguments: A relation.

  Behaviour: It adds relation R to the current intermediate result.

- **Rel_del** (R)

  Arguments: A relation.

  Behaviour: It eliminates relation R from the current intermediate result.

Operations over a relation:

- **Att_add** (X, R)

  Arguments:
  - A set of attributes.
  - A relation.

  Behaviour: It adds a set of attributes to a relation. If the relation does not exist it generates a new one.

- **Att_rem** (X, R)

  Arguments:
  - A set of attributes.
  - A relation.

  Behaviour: It removes a set of attributes from a relation.

- **Att_cpy** (X, R, R')

  Arguments:
  - A set of attributes.
  - A relation. This is the origin of the copy.
  - A relation. This is the target of the copy.

  Behaviour: It copies a set of attributes from one relation to another. If the relation does not exist it generates a new one.

- **Att_calc** (X, f, Y, A)

  Arguments:
  - A set of attributes.

    The attributes that participate in the calculation function of the derived attribute.
  - A calculation function.
  - A set of attributes.

    The attributes that are required for the calculation of the derived attribute.
  - An attribute name.

    The name for the derived attribute, including the name of the relation.

  Behaviour: It adds a derived attribute to a relation.

Note: The attributes *required* for a calculation are those without which the calculation cannot be done.

Attributes that are included in the “group by” clause of the SQL query used for the data load of a calculated attribute, are not considered as required. This is because if one of these attributes is removed the only thing we have to do is the re-calculation of the instance. Therefore it is not necessary to maintain the information of this attribute “dependency” in the trace. We are interested only in “schema dependencies” and we assume that every time a modification is made to the trace, the corresponding data loading processes are re-generated.

Operations over a set of keys of a relation:

- **Key_add** (X, $Att_K(R)$)

  Arguments: - A set of attributes.
  - The set of keys of a relation.

  Behaviour: It adds a key to a set of keys. If the set of keys does not exist it generates a new one.

- **Key_del** (X, $Att_K(R)$)

  Arguments: - A set of attributes.
  - The set of keys of a relation.

  Behaviour: It eliminates a key from a set of keys.

- **FKey_add** (X, $Att_{FK}(R_1,R_2)$, $Att_{FK}(R_1)$)

  Arguments: - A set of attributes.
  - The foreign key between relations R_1 y R_2.
  - The set of foreign keys of relation R_1.

  Behaviour: It generates and adds a foreign key to a set of foreign keys.

- **FKey_del** ($Att_{FK}(R_1, R_2)$, $Att_{FK}(R_1)$)

  Arguments: - The foreign key between relations R_1 y R_2.
  - The set of foreign keys of relation R_1.

  Behaviour: It eliminates a foreign key from a set of foreign keys.

3. Appendix 3 - The Primitives in terms of Basic Operations

Primitive 1. IDENTITY

Description:

It adds a relation that is the same as the source one. It removes the source one.

Input:

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

Basic Operations:

- Rel_add (R’)
- Rel_del (R)

Primitive 2. DATA FILTER

Description:

It adds a relation that is the same as the source one, except for a set of attributes that are removed. It removes the source relation.

Input:

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

Basic Operations:

- Att_cpy ({*Att(R)*-X}, R, R’)
- Rel_add (R’)
- Rel_del (R)

Primitive 3. TEMPORALIZATION

Description:

It adds a relation that is the same as the source one, except for an attribute that is included in the relation. Optionally, it extends the key of the relation with the new attribute. It removes the source relation.

Input:

- source schema : R (A_1, ..., A_n) / $\exists$ X $\subset$ { A_1, ..., A_n } $\wedge$ X $\in$ *$Att_K(R)$*
- T time attribute / DOM(T) = { c / c $\subseteq$ { t_0, ..., t_k } set of time measures } , or

  DOM(T) = { 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

Basic Operations:

- Att_cpy (*Att(R)*, R, R')
- Att_add ({T}, R')
- if key then Key_add (XT, *$Att_K(R')$*)
- Rel_add (R')
- Rel_del (R)

Primitive 4. GROUP: KEY GENERALIZATION

Primitive 14.1. VERSION DIGITS

Description:

It adds a relation that is the result of substituting the key attribute by another one in the source relation. It removes the source relation.

Input:

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

Basic Operations:

- Att_cpy ({*Att(R)*-A_1}, R, R')
- Att_add ({B}, R')
- Key_add ({B}, *$Att_K(R')$*)
- Rel_add (R')
- Rel_del (R)

Primitive 14.2. KEY EXTENSION

Description:

It adds a relation that is the same as the source one, except for its key which is extended with a set of attributes. It removes the source relation.

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

Basic Operations:

- Att_cpy (*Att(R)*, R, R')
- Key_add (XY, $Att_K(R')$)
- Rel_add (R')
- Rel_del (R)

Primitive 5. FOREIGN KEY UPDATE

Description:

It adds a relation that is the result of substituting a set of attributes by another one in the source relation (the set of attributes eliminated are a foreign key in the source relation). It also defines the new set of attributes as foreign key with respect to a set of relations. It removes the source 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

Basic Operations:

- Att_cpy ({*Att(R)*-X}, R, R')
- Att_add (Y, R')
- FKey_add (Y, $Att_{FK}(R',R_i)$, $Att_{FK}(R')$), i: 1..m
- Rel_add (R')
- Rel_del (R)

Primitive 6. GROUP: DD-ADDING

Primitive 14.3. DD-ADDING 1-1

Description:

It adds a relation that is the same as the source one, except for a new attribute that is calculated from others of the same relation. It removes the source 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

Basic Operations:

- Att_cpy (*Att(R)*, R, R')
- Att_calc ({R.A_{i1}, ..., R.A_{im}}, f, ∅, R'.A_{n+1})
- Rel_add (R')
- Rel_del (R)

Primitive 14.4. DD-ADDING N-1

Description:

It adds a relation that is the same as the source one, except for a new attribute that is calculated from attributes that belong to another relation. It removes the source relations.

Input:

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

Basic Operations:

- Att_cpy (*Att(R_1)*, R_1, R'_1)
- Att_calc (R_1.X ∪ R_2.Y, f, {A}, R'_1.A_{n+1}) , where X ∪ Y = (C_1, ..., C_k)
- if is_fk then FKey_add ({A_{n+1}}, *Att_{FK}(R'_1,R_3)*, *Att_{FK}(R'_1)*)
- Rel_add (R'_1)
- Rel_del (R_1)

- Rel_del (R_2)

Primitive 14.5. DD-ADDING N-N

Description:

It adds a relation that is the same as the source one, except for a new attribute that is calculated from an attribute of another relation, through an aggregate operation. It removes the source relations.

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

Basic Operations:

- Att_cpy (*Att(R_1)*, R_1, R'_1)
- Att_calc ({R_2.B}, e, {R_2.A}, $R'_1.A_{n+1}$)
- Rel_add (R'_1)
- Rel_del (R_1)
- Rel_del (R_2)

Note: There are some attributes of R_1 that could affect the results of the calculation. These attributes are the ones used in the "group by" clause of the SQL query that must be executed for loading the resulting relation schema. We consider these attributes as "not required" for the calculation because in case one of them is removed, the calculation still can be done, although perhaps corresponding data need to be re-loaded. We find the same situation in Primitives P8 and P9.

Primitive 7. ATTRIBUTE ADDING

Description:

It adds a relation that is the same as the source one, except for a set of attributes that are included in it. It removes the source relation.

Input:

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

Basic Operations:

- Att_cpy (*Att(R)*, R, R')
- Att_add ({B_1, ..., B_m}, R')
- Rel_add (R')
- Rel_del (R)

Primitive 8. HIERARCHY ROLL UP

Description:

It adds a relation that is the result of substituting a set of attributes by one attribute in the source relation R_1, and eliminating the foreign key defined from R_1 to R_2. It removes this source relation. Optionally, it adds a relation that is the same as R_2 except for a set of attributes that are eliminated, and it defines a primary key for R_2 and a foreign key from R_1 to R_2. Finally, it removes R_2.

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

Basic Operations:

- Att_cpy (*Att(R_1)*, R_1, R'_1)
- Att_calc (R_2.B, =, {R_2.A, R'_1.A}, R'_1.B)
- Att_rem ({A}, R'_1)
- for i: 1..k do

    Att_calc ({R_1.Z_i}, e_i, $\emptyset$, R'_1.Z'_i)

    Att_rem ({Z_i}, R'_1)
- FKey_del ($Att_{FK}(R'_1, R_2)$, $Att_{FK}(R'_1)$)
- Rel_add (R'_1)
- Rel_del (R_1)
- if agg_h then
    - Att_cpy ({Att(R_2)-Y}, R_2, R'_2)
    - Key_add ({B}, $Att_K(R'_2)$)

- FKey_add ({B}, $Att_{FK}(R'_1, R'_2)$, $Att_{FK}(R'_1)$)
- Rel_add (R'_2)

▪ Rel_del (R_2)

Primitive 9. AGGREGATE GENERATION

Description:

It adds a relation that is the result of substituting a set of attributes X by another set Y in the source relation. Each attribute of Y is calculated from an attribute of X. It removes the source relation.

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

Basic Operations:

```
▪ Att_cpy ({Att(R)-Y}, R, R')
▪ for i: 1..k do
        Att_calc ({R'.Z_i}, e_i, ∅, R'.Z'_i)
        Att_rem ({Z_i}, R')
▪ Rel_add (R')
▪ Rel_del (R)
```

Primitive 10. DATA ARRAY CREATION

Description:

It adds a relation that is the result of substituting a set of attributes X by another set Y in the source relation. Each attribute of Y is calculated from an attribute of X. It removes the source relation.

Input:

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

Basic Operations:

- Att_cpy ({*Att(R)*-{A, B}}, R, R')
- Att_calc ({R.A}, =, {R.B}, R'.V_1)
- ………………
- Att_calc ({R.A}, =, {R.B}, R'.V_k)
- Rel_add (R')
- Rel_del (R)

Primitive 11. GROUP: PARTITION BY STABILITY

Source schema:

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

Primitive 14.6. VERTICAL PARTITION

Description:

It adds three relations, each of one is the result of removing a set of attributes from the source relation. It removes the source relation.

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

Basic Operations:

- Att_cpy ({*Att(R)*-{Z$\cup$W}}, R, R_1)
- Att_cpy ({*Att(R)*-{Y$\cup$W}}, R, R_2)
- Att_cpy ({*Att(R)*-{Y$\cup$Z}}, R, R_3)
- Rel_add (R_1)
- Rel_add (R_2)
- Rel_add (R_3)
- Rel_del (R)

Primitive 14.7. HORIZONTAL PARTITION

Description:

It adds two relations that are the same as the source one. It removes the source relation.

Input:

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

Basic Operations:

- Att_cpy (*Att(R)*, R, R_{Cur})
- Att_cpy (*Att(R)*, R, R_{His})
- Rel_add (R_{Cur})
- Rel_add (R_{His})
- Rel_del (R)

Primitive 12. GROUP: HIERARCHY GENERATION

Source schema:

- R_1, ..., R_n / $\exists$ A / A $\in$ $Att_D(R_i)$, i= 1...n $\wedge$ A is the lowest level of one hierarchy

Primitive 14.8. DE-NORMALIZED HIERARCHY GENERATION

Description:

It adds a relation R' constructed with attributes that are given in the input (J_1, ..., J_m) and defines a key for it. Besides it adds a set of relations that are the same as the input ones (R_1, …., R_n), except for some attributes (J_1, ..., J_m) that are removed from them and one attribute (K) that is included in them. This attribute is defined as foreign key with respect to R', in each of these relations. The source relations are removed.

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

Basic Operations:

- Att_add ({J_1, ..., J_m}, R')
- Key_add ({K}, $Att_K(R')$)
- Att_cpy ({$Att(R_i)$- {J_1, ..., J_m}}, R'_i) i: 1..n
- Att_add ({K}, R'_i) i: 1..n
- FKey_add ({K}, $Att_{FK}(R'_i, R')$, $Att_{FK}(R'_i)$) i: 1..n
- Rel_add (R')

- Rel_add (R'_i) i: 1..n
- Rel_del (R_i) i: 1..n

Primitive 14.9. SNOWFLAKE HIERARCHY GENERATION

Description:

It adds a set of relations (R_{J1}, …., R_{Jm-1}) constructed with attributes that are given in the input (J_1, ..., J_m) and defines a key and a foreign key for each of them. Besides it adds a set of relations that are the same as the input ones (R_1, …., R_n), except for some attributes (J_1, ..., J_m) that are removed from them and one attribute (K) that is included in them. This attribute is defined as foreign key with respect to R_{J1}, in each of these relations. The source relations are removed.

Input:

- source schema : the source schema defined for the group
- J_1, ..., J_m , sorted list of attributes that constitutes a hierarchy /

$$A \in \{ J_1, J_2 \} \wedge \text{ A is the lowest level}$$

- K / $K \in \{ J_1, ..., J_m \}$ key for the hierarchy
- source instance : r_1, ..., r_n

Basic Operations:

- Att_add ({J_i, J_{i+1}}, R_{Ji}) i: 1..m-1
- Key_add ({J_i}, $Att_K(R_{Ji})$) , i: 1..m-1
- FKey_add ({J_{i+1}}, $Att_{FK}(R_{Ji}, R_{Ji+1})$, $Att_{FK}(R_{Ji})$) , i: 1..m-1
- Att_cpy ({$Att(R_i)$-{J_1, ..., J_m}}, R_i, R'_i) , i: 1..n
- Att_add ({K}, R'_i) , i: 1..n
- FKey_add ({K}, $Att_{FK}(R'_i, R_{J1})$, $Att_{FK}(R'_i)$) , i: 1..n
- Rel_add (R_{Ji}) , i: 1..m-1
- Rel_add (R'_i) , i: 1..n
- Rel_del (R_i) , i: 1..n

Primitive 14.10. FREE DECOMPOSITION - HIERARCHY GENERATION

Description:

It adds a set of relations (R_{J1}, …., R_{Jm-1}) constructed with attributes that are given in the input (J_1, ..., J_m) and defines a key for each of them. Besides it adds a set of relations that are the same as the input ones (R_1, …., R_n), except for some attributes (J_1, ..., J_m) that are removed from them and one attribute (K) that is included in them. This attribute is defined as foreign key with respect to R_{J1}, in each of these relations. The source relations are removed.

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

Basic Operations:

- Att_add (X_i, R_{Ji}) , i: 1..h
- Key_add ({K}, $\mathit{Att}_K(R_{J1})$)
- Att_cpy ({$\mathit{Att}(R_i)$-{J_1, ..., J_m}}, R_i, R'_i) , i: 1..n
- Att_add ({K}, R'_i) , i: 1..n
- FKey_add ({K}, $\mathit{Att}_{FK}(R'_i, R_{J1})$, $\mathit{Att}_{FK}(R'_i)$) , i: 1..n
- Rel_add (R_{Ji}) , i: 1..m-1
- Rel_add (R'_i) , i: 1..n
- Rel_del (R_i) , i: 1..n

Primitive 13. MINIDIMENSION BREAK-OFF

Description:

It adds two relations R_1 and R_2. R_1 is the result of substituting in R, a set of attributes X by one attribute K, and defining this attribute as foreign key to R_2. R_2 is a new relation which attributes are the set of attributes X, and K is defined as its key. It removes the source relation R.

Input:

- source schema : R (A_1, ..., A_n) $\in \mathit{Rel}_D$
- K , key for the new dimension
- $X \subset \{ A_1, ..., A_n \}$, set of attributes of the minidimension
- source instance : r

Basic Operations:

- Att_cpy ({$\mathit{Att}(R)$-X}, R, R_1)
- Att_add ({K}, R_1)
- FKey_add ({K}, $\mathit{Att}_{FK}(R_1, R_2)$, $\mathit{Att}_{FK}(R_1)$)
- Att_add ({K}$\cup$X, R_2)
- Key_add ({K}, $\mathit{Att}_K(R_2)$)
- Rel_add (R_1)

- Rel_add (R_2)
- Rel_del (R)

Primitive 14. NEW DIMENSION CROSSING

Description:

It adds a new relation, which attributes are the union of the attributes of all the source relations minus the attributes specified in the input to be excluded. Besides, it defines the key of the new relation as the union of the keys of all the source relations. It removes the source relations.

Input:

- source schema : $R_1, ..., R_n$ / $R_i \in (Rel_M \cup Rel_D)$, $i = 1..n \wedge$
  $Att_K(R_i) = X_i \wedge$
  $R_j \cap R_{j+1} = A_j$, $j = 1..n-1$
- $Y_1, ..., Y_n$, sets of attributes to be excluded from the resulting relation / $X_i \not\subset Y_i$
- source instance : $r_1, ..., r_n$

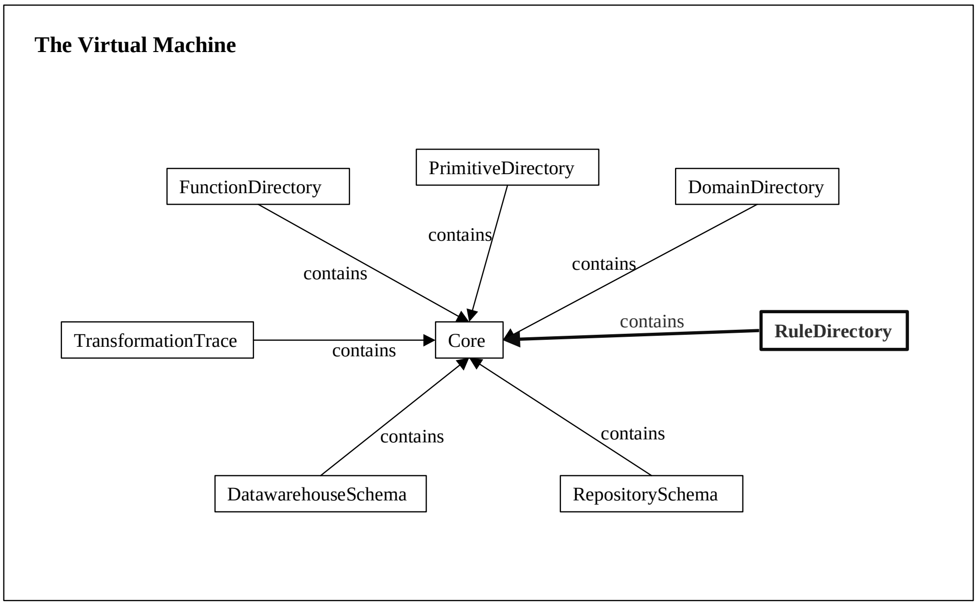
Basic Operations:

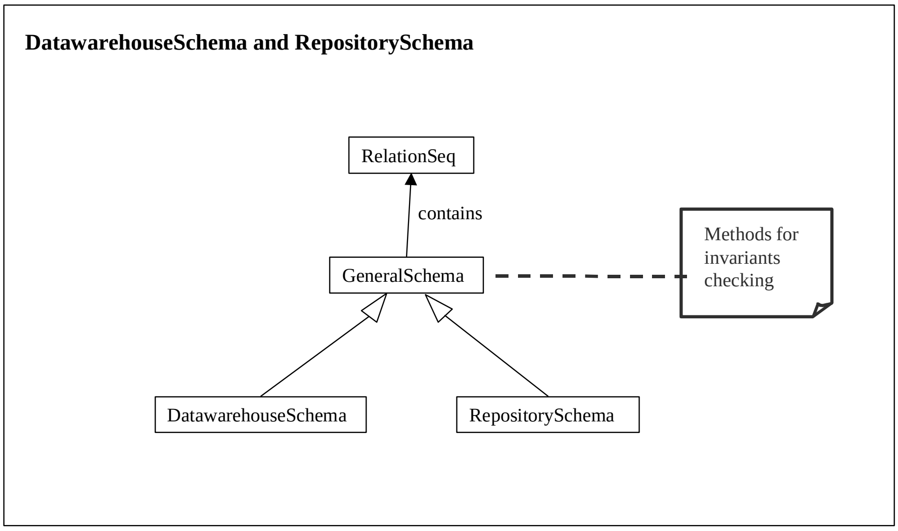
- Att_add ($\cup_{i=1..n}$ ($Att(R_i)$ – Y_i), R)
- Key_add ($\cup_{i=1..n}$ X_i, $Att_K(R)$)
- FKey_add (X_i, $Att_{FK}(R,R_i)$, $Att_{FK}(R)$), i: 1..n
- Rel_add (R)
- Rel_del (R_i) , i = 1..n

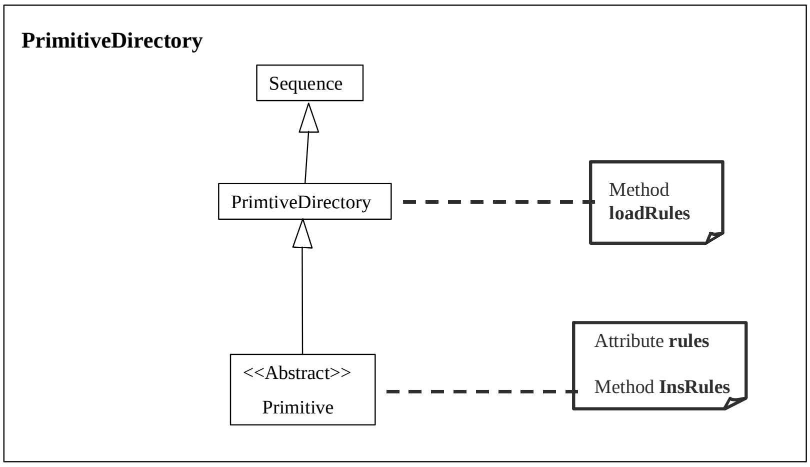
4. Appendix 4 - Class Diagram of the DW Design Tool

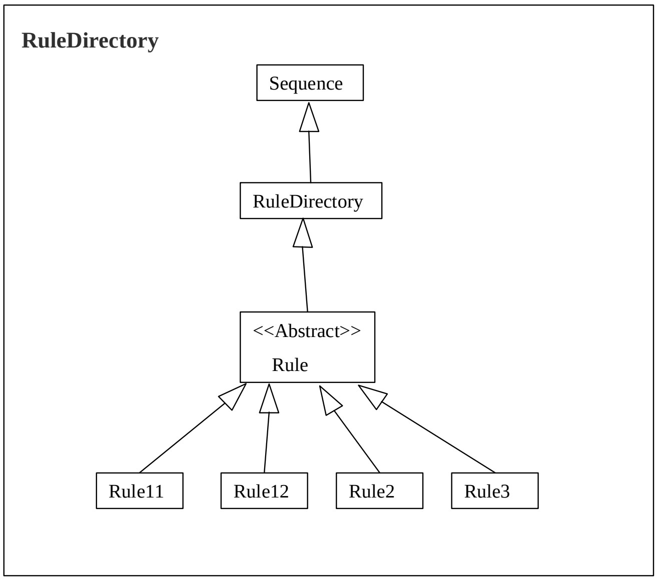
In this appendix we show the class diagram of highest level, and the class diagrams of lower levels that are necessary to show the parts that were added in the context of the present thesis, i.e. the representation of the Invariants and the Rules. The rest of the class diagram can be found in [Gar99].

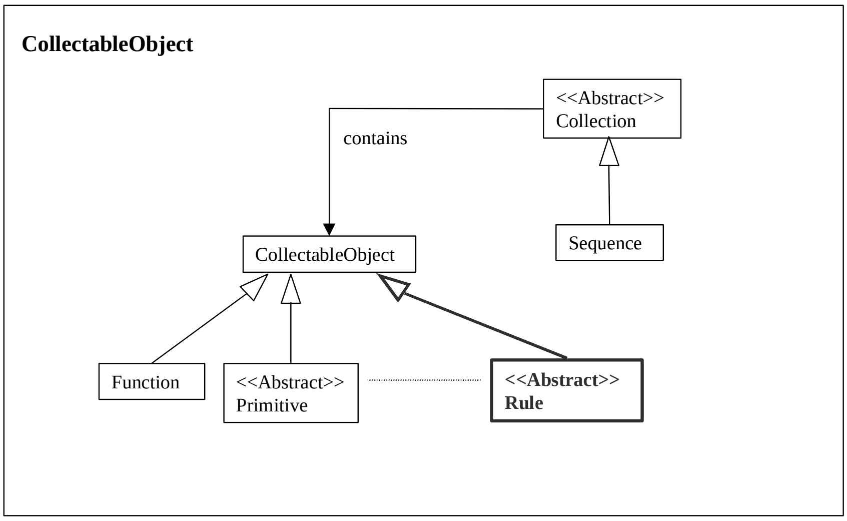
The parts of the diagram that are marked in bold and the RuleDirectory diagram, are the parts that were added in the present work.




DatawarehouseSchema and RepositorySchema
RelationSeq
contains
GeneralSchema
Methods for invariants checking
DatawarehouseSchema
RepositorySchema


PrimitiveDirectory
Sequence
PrimtiveDirectory
Method
loadRules
Attribute rules
Method InsRules
<<Abstract>>
Primitive


RuleDirectory
Sequence
RuleDirectory
<<Abstract>>
Rule
Rule11
Rule12
Rule2
Rule3


CollectableObject
<<Abstract>>
Collection
contains
Sequence
CollectableObject
Function
<<Abstract>>
Primitive
<<Abstract>>
Rule

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.